Disk Inside

Disk Inside

source code and executables for all articles

# AMIGA WORLD TECH JOURNAL

## ARTICLES


2 **Amiga NTSC and PAL Genlock Interfacing** *By Scott Hood*
Make devices meet the Amiga's video standards

6 **A Tight Fit: JPEG Compression** *By Perry Kivolowitz*
Squeeze your graphics files

10 **Blitter Optimization** *By Leo L. Schwab*
Use one bitplane, move many

26 **ARexx is for Writing Applications, Too** *By Marvin Weinstein*
This calendar program is (dated) proof

30 **Trouble-Free Data Sharing** *By Eric Giguere*
Four methods of data arbitration

38 **An Intuition Shortcut: EzLib** *By Dominic Giampaolo*
A library of simple GUI routines

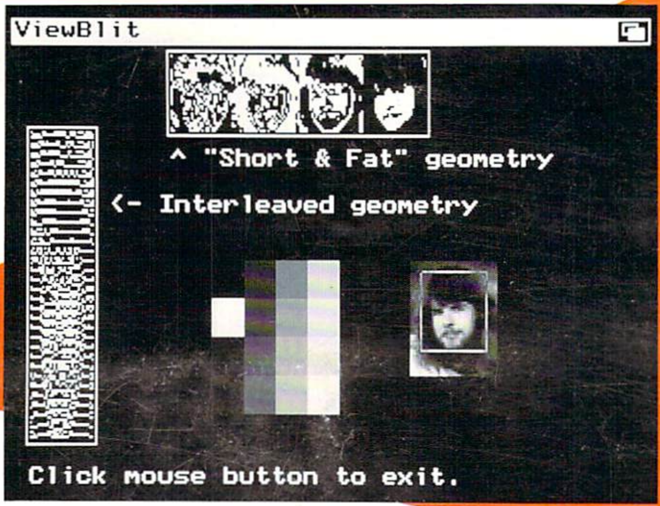
42 **Designing the User Interface: Proper Dragging** *By David "Talin" Joiner*
How should your program react to movements?

45 **MIDI Programming Demystified** *By Darius Taghavy*
Output to MIDI devices

50 **A Developer's Guide to PostScript—Part I** *By Tomas Rokicki*
Language conventions and basic commands

59 **Advanced Use of the Audio Device** *By David "Talin" Joiner and Joe Pearce*
Pooling and looping are the keys

62 **Better Bezier Curves** *By Robert Wittner*
Try successive linear approximations


Fool the Blitter with special geometry; find out how on p. 10.

## REVIEWS


36 **Intuition: A Practical Programmer's Guide** *By John Toebes*
Read along with the code


## COLUMNS


1 **Message Port**
Where are we going?

14 **Digging Deep in the OS** *By Paul Miller*
Using graphics.library's primitives

20 **Graphics Handler** *By Brian Carmichael*
Faster 3-D display routines

32 **TNT**
The pick of the new products

64 **Letters**
Questions, questions, questions


## ON DISK

*(See page 33.)*

**Tons of Tools for Programmers**

**JPEG Compression Routines**

***Plus source code and executables for articles***


0 74470 12070 4 02

*U.S.A. $15.95*
*Canada $19.95*

# SAVE IT. MOVE IT. GET IT BACK.

**Back-Up...Transfer...Retrieve Quickly And Easily With Central Coast's Software For The Amiga**

Valuable utility programs can save you time, money and, in the case of catastrophic errors like hard drive failure, possibly months of work.

## Quarterback Tools – Recover Lost Files

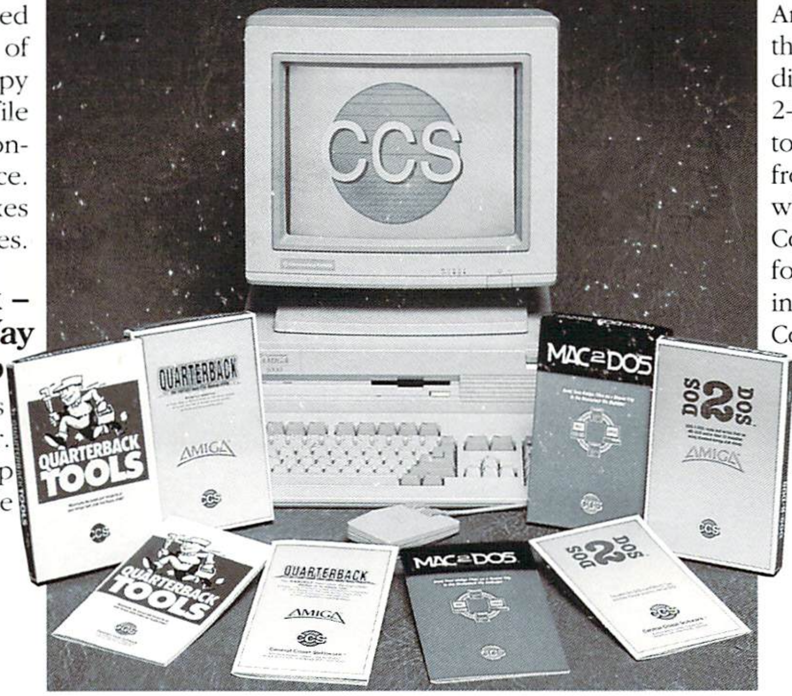
Fast and easy. Reformats all types of disks – either new or old filing systems – new or old Workbench versions. Also optimizes the speed and reliability of both hard and floppy disks. Eliminates file fragmentation. Consolidates disk space. Finds and fixes corrupted directories.

## Quarterback – The Fastest Way To Back-Up

Backing-up has never been easier. Or faster. Back-up to, or restore from: floppy disks, streaming tape (AmigaDOS-compatible), Inner-Connection's Bernoulli drive, or ANY AmigaDOS-compatible device.

## Mac-2-Dos & Dos-2-Dos – A Moving Experience

It's easy. Transfer MS-DOS and ATARI ST text and data files to-and-from AmigaDOS using the Amiga's own disk drive with Dos-2-Dos; and Macintosh files to-and-from your Amiga with Mac-2-Dos. Conversion options for Mac-2-Dos include ACSII, No Conversion, MacBinary, Postscript, and MacPaint to-and-from IFF file format.




**Central Coast Software**

**A Division Of New Horizons Software, Inc.**


206 Wild Basin Road, Suite 109, Austin, Texas 78746
(512) 328-6650 • Fax (512) 328-1925




Circle 3 on Reader Service card.

*The AmigaWorld Tech Journal* (ISSN 1054-4631) is an indepedent journal not connected with Commodore Business Machines, Inc. *The AmigaWorld Tech Journal* is published bi-monthly by IDG Communications/Peterborough, Inc., 80 Elm St., Peterborough, NH 03458. U.S. Subscription rate is $69.95, Canada and Mexico $79.95, Foreign Surface $89.95, Foreign Airmail $109.95. All prices are for one year. Prepayment is required in U.S. funds drawn on a U.S. bank. Application to mail at 2nd class postage rates is pending at Peterborough, NH and additional mailing offices. Phone: 603-924-0100.  **Postmaster**: Send address changes to *The AmigaWorld Tech Journal*, 80 Elm St., Peterborough, NH 03458. *The AmigaWorld Tech Journal* makes every effort to assure the accuracy of articles, listings and circuits published in the magazine. *The AmigaWorld Tech Journal* assumes no responsibility for damages due to errors or omissions. Third class mail enclosed.

Bulk Rate
US Postage Paid
IDG Communications/Peterborough, Inc.

# MESSAGE PORT

*Stay tuned…*

So, after a year of *Tech Journal*s, where are we and where are we going?

Physically, we're at 64 pages; that's up 16 pages since the first issue. Our publisher promises we'll continue to grow, but when and how much remains to be seen. Despite all the requests we've received, we have no plans, as yet, to go monthly. In other words, you won't get the *Tech Journal* more often, but it may be fatter when you get it.

What do we intend to put on all these pages? A new column for one thing. In response to the pleas we've received not to forget the beginners, next issue we're starting a (currently unnamed) column to cover introductory *Amiga* programming concepts in BASIC and C. Don't worry, we're not going back on our promise of "no three-part dissertations on C's printf() command." We're just committing a column and at least one article per issue to subjects such as setting up screens, windows, menus, gadgets, message ports, and more. We hope this will give new users the background they need to tackle the more advanced topics.

Speaking of which, I reread the list of promised topics in my first editorial. Of those 14, we've covered nine so far (and a tenth was traded out to *AmigaWorld*)—not a bad batting average (.643 to be exact). The remaining four were edged out by a landslide of ideas from authors, peer reviewers, and, most importantly, readers. Considering the events of September's DevCon—both Commodore's future plans and the abundance of ideas flung at me—you've got a lot of interesting articles to look forward to. Some topics on the docket are the ASL requester, Boopsi, programming TIGA graphic devices, interface issues, game design, animation routines, and, of course, anything else you suggest.

Starting with this issue, you'll see we made a few changes you called for to the *Tech Journal* disk, as well. The most obvious is we've dropped the parentheses in the directory names. (Did I hear a cheer?) We've also added two new features: a doc file for lharc, because some of you are having trouble decompressing files, and an on-going index of *Tech Journal* articles and disk contents. We'll continue to fill any leftover space with utilities, libraries, and languages.

If you have a program or library you're particularly proud of that might help your fellow programmers, send it along. If we like it (and have room), we'll put it on the disk. Keep in mind, however, space is limited. If your utility consumes half a disk or more, chances are we won't have enough space.

If there's one underlying direction in where we're going, it's that we're following your lead and Commodore's rules. We try to cover the subjects you want to learn about, while teaching techniques that ensure your programs to work with everyone elses and future generations of the Amiga—generations we're looking forward to covering. ■

Linda JB Laflamme

# Amiga NTSC and PAL Genlock Interfacing

*Your genlock will be useless if it doesn't meet the Amiga's expectations. Take a look at what they are.*

**By Scott Hood**

BUILDING A GENLOCK requires more than a schematic and a trip to Radio Shack for the parts. You must understand how the device addresses both external video equipment and the Amiga to ensure you have a top-notch design. This article details the interfacing information you need to develop both NTSC- and PAL-based genlocks. A thorough knowledge and understanding of digital logic design principles and composite video signals in general is therefore a prerequisite for understanding the details presented in this article. For some general background on Amiga genlocks, consult "Improved Genlock Handling"(p. 27, June/July '91).

### GENLOCK OPERATION

For a point of reference, you should consider *what* genlocks do before getting caught up in the details of *how* they work. Genlocks designed for the Amiga synchronize the Amiga's video-beam counters and timing to an external video source and overlay the Amiga graphics data onto the external video source for output. In its most basic form this is done by providing the Amiga's graphics hardware (the Agnus VLSI IC) with horizontal (*HSYNC) and vertical (*VSYNC) sync reset signals that are generated from the genlock itself and derived from the external video source. This also requires that the genlock generate from the external video source a master dot-clock signal (XCLK) that clocks the Amiga's graphics hardware.

Once the Amiga's video output (analog RGB at 0.7Vp-p/75 ohm) has been synchronized to the external video source, the genlock can overlay the Amiga's graphics data onto the external video data under the control of the Amiga's pixel switch signal (*PIXELSW or *ZD). The Amiga uses this pixel-switch signal as a transparency control that, when asserted, indicates that the genlock should output the external video source for that specified pixel location. The pixel-switch signal can be programmed to represent either an entire bitplane of transparency bits to be used as a bitplane mask or any of the Amiga's color values (assuming the Amiga system has the ECS or later graphics chips).

### AMIGA SIGNALS

With the basics out of the way, let's look in detail at all of the signals used in interfacing a genlock to the Amiga. (For the video-slot pinout and the DB23 video port pin assignments, see Commodore's *A500/A2000 Technical Reference Guide.*)

The following is a list of the signals handled through the internal video slot on A2000s, A3000s, and A3000Ts, as well as through the external 23-pin (DB23) video port in all Amigas. An * prefix on a signal name indicates the signal is active low.

• *HSYNC (Input, TTL level): A genlock uses the horizontal-sync-reset signal to tell the Amiga that the start of a line sequence has occurred. The signal is clocked into Agnus on the rising edge of the *C1 clock and must meet 30ns of setup time and 30ns of hold time relative to this clock. (More on this later.)

• *VSYNC (Input, TTL level): A genlock uses the vertical-sync-reset signal to inform the Amiga that the start of a frame sequence has occurred. Clocked into Agnus on the rising edge of the *C1 clock, *VSYNC must meet 30ns of setup time and 30ns of hold time relative to *C1. (More details will follow.)

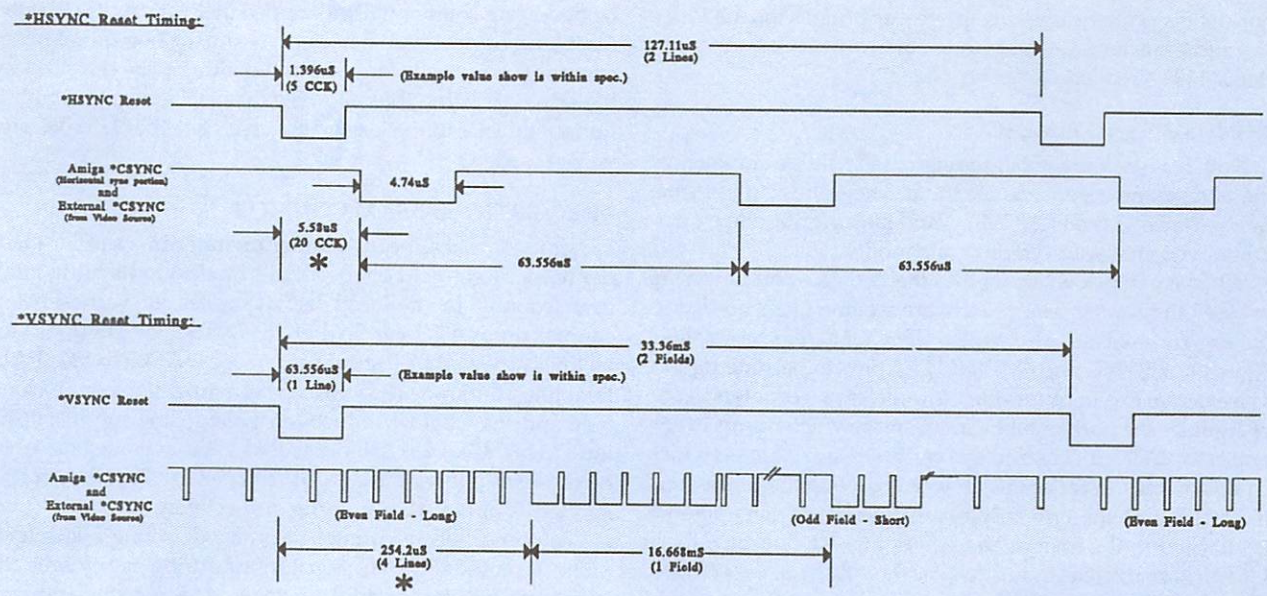
• XCLK (Input, TTL level): This input is the 28.63636 MHz (NTSC) or 28.375156MHz (PAL) master dot-clock signal that the genlock must generate as part of the synchronizing process. It is a 50-percent duty-cycle clock that must be relatively jitter free and fairly clean with little or no undershoot or overshoot.

• *XCLKEN (Input, TTL): A genlock must drive this input low. The *XCLKEN signal forces the Amiga's hardware to use the XCLK input clock, rather than the internal oscillator, as the master dot clock.

• *PIXELSW (Output, TTL): *PIXELSW, a buffered output signal, tells the genlock device when to assert the external video source as the output for a given pixel. Thus if the signal is disasserted (a logic 1 or high), the genlock is to output the Amiga's RGB graphics data for that pixel. This signal is sometime also called the "Zero-detect" or *ZD signal for historic reasons. This output has no fixed phase relationship to any of the system clocks (*C1 or XCLK), but is correctly timed to correspond with the pixel data.

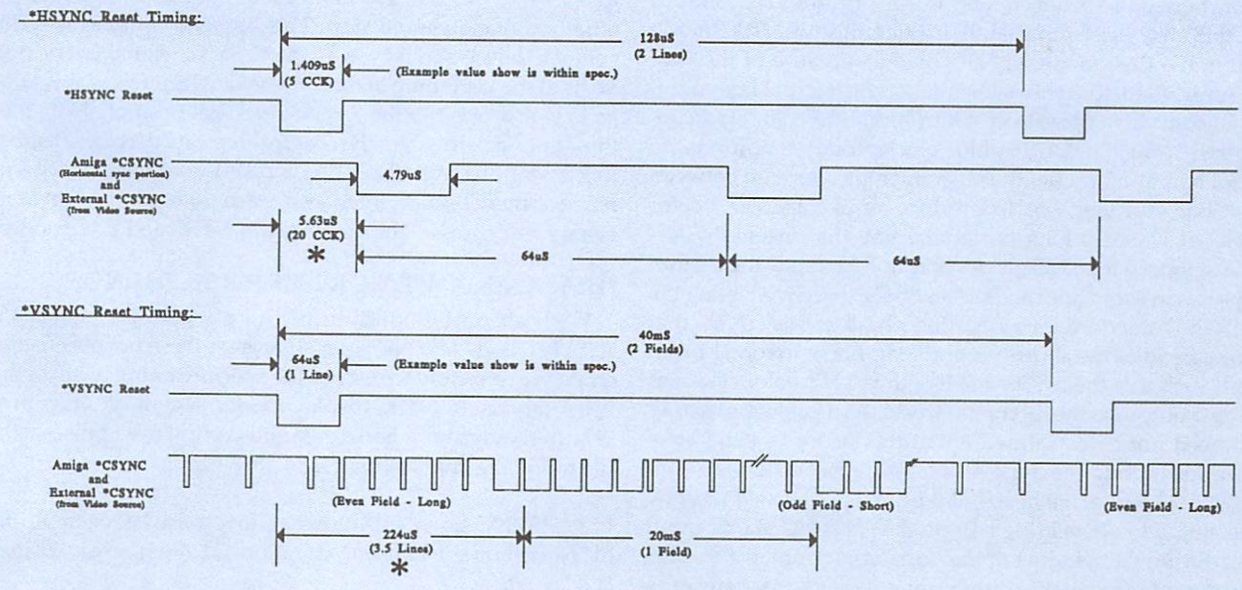
• *C1 (Output, TTL): This buffered output clock (sometimes called the CCK clock) is the dot clock divided by eight that internally qualifies (on the rising edge) the genlock generated *HSYNC and *VSYNC sync-reset signals to Agnus. It is a 50-percent duty-cycle clock that bears no fixed phase relationship to the XCLK edges. In NTSC systems its frequency is 3.579545 MHz and in PAL systems, 3.5468945 MHz.

• Analog RED, GREEN, and BLUE (Outputs, 0.7Vp-p/75 ohm): These are the Amiga's separate analog RGB outputs that a genlock uses as overlay data. The minimum analog pixel width is 35ns with the ECS or later chip set. These outputs should be 75-ohm loaded and capacitively coupled into the




* Note: These values are fixed and reflect the Agnus Beam counter reset positions.

## Figure 1. NTSC *HSYNC and *VSYNC Reset Timing




* Note: These values are fixed and reflect the Agnus Beam counter reset positions.

## Figure 2. PAL *HSYNC and *VSYNC Reset Timing

genlock's overlay circuitry. They have no fixed phase relationship to any of the system clocks (*C1 or XCLK).

• *CSYNC (Output, TTL): This buffered output signal is the system's composite-sync reference signal. Many genlocks use this composite sync signal to run their on-board composite encoder circuits. When the genlock and the Amiga are correctly synchronized, the decoded composite sync from the external video source and the Amiga's *CSYNC signal will match or be in sync.

• +12V, +5V, –5V, Gnd (Outputs, current limited): These are the Amiga-provided power lines that genlocks are allowed to use. Always make sure to connect the genlock's and Amiga's ground lines together for proper grounding. The power budget of the power lines is fairly tight; designers, pay at- ►

tention to your use of current provided from these supplies. For details on the difference in power budget for the video slot and the external 23-pin video port, consult Commodore's *A500/A2000 Technical Reference Guide.*

## AGNUS'S SYNC TIMING

Now that we know what the signals are, let's examine how the Amiga and a genlock use them. The following information pertains to the ECS (8372x) and later Agnus chips operating in normal scan (15KHz) rate modes.

When a genlock is attached to the Amiga upon power up or reset, the Amiga system software sets the ERSY bit (External ReSYnc enabled, bit 1) in the BPLCON0 register to allow Agnus's *HSYNC and *VSYNC I/O pins to become inputs. The external resync inputs therefore affect two counters inside of Agnus—the horizontal-beam counter, which counts in color clocks (CCK clocks), and the vertical-beam counter, which counts in lines. Because the genlock is driving these pins (and the XCLK pin), the system software determines that a genlock is attached to the system and leaves this bit, and thus these I/O pins, enabled as inputs to sync the Amiga to the genlock. The system software then sets a system-wide flag that programs can test to determine that a genlock is present. OS 2.0 also uses this information to adjust the graphics database so that the user can select only the appropriate genlockable display modes in the Preferences ScreenMode program.

## THE HORIZONTAL-BEAM COUNTER

The horizontal-beam counter inside of Agnus responds to the *HSYNC reset input signal differently in NTSC mode than in the PAL mode. In NTSC mode, because of the way the Amiga's custom chips generate the horizontal line count, the custom chips operate in "short line" (227 color clocks total) and "long line" (228 color clocks total) conditions. In NTSC operating mode, the custom chips alternate between short line and long line to produce an average 227.5 color clock (CCK clock) long horizontal line that meets the RS-170A standard for 63.556µs per line. In PAL mode the custom chips run in short-line mode at all times, thereby meeting the CCIR EBU standard for a PAL line length of 64µs. (Note that Denise handles the short-line and long-line horizontal-beam count such that the NTSC line length is 227.5 color clocks as the data output to the rest of the world. As a genlock designer you need not be concerned about this or try to generate a short-line or long-line type of *HSYNC reset signal.)

In NTSC mode, the horizontal-beam counter will reset by detecting a low level (logic 0) on the *HSYNC signal input only during the window of the horizontal count of $00. This may sound strange, but the beam counter's mechanism is such that the counter stops at the beginning of a short line (holds at count $00) if the reset level has not been detected. Because the Agnus horizontal-beam counter is essentially stopped and waiting to be resynced by the external source, the mechanism allows horizontal locking to the external source once the *HSYNC signal is asserted, causing the counter (now sync locked to the external source) to resume. In NTSC modes, the resumed horizontal-beam counter then counts out a short line that is followed by a long line before the next *HSYNC reset is needed. This therefore implies that the genlock device should only assert the *HSYNC signal once every two lines in NTSC mode.

In PAL mode, the horizontal-beam counter is always counting out short lines and requires that the *HSYNC reset signal be asserted every line. As in the NTSC modes, the horizontal-beam counter will be stopped (held at count $00) if the reset level has not been detected. Again, the horizontal-beam counter is essentially stopped and waiting to be resynced by the external source, thus allowing the mechanism to establish horizontal lock to the external source once the *HSYNC signal is asserted.

## THE VERTICAL-BEAM COUNTER

Agnus's vertical-beam counter operates in a similar manner (either long-field or short-field modes) to its horizontal-beam counter. In interlaced display modes, the vertical-beam counter outputs a long field of 263 (NTSC) or 313 (PAL) total lines followed by a short field of 262 (NTSC) or 312 (PAL) total lines, and repeats. Agnus, alternating between the long field and short field, produces an average line count of 262.5 lines in NTSC and 312.5 lines in PAL. This average line count mechanism allows the output to meet the RS-170A NTSC and CCIR EBU PAL standards, respectively.

The vertical-beam counter resets by detecting a low level (logic 0) on the *VSYNC signal input during at any time the vertical count. This counter, however, does not stop and wait to be reset as the horizontal-beam counter does. The reset happens such that the next line of the the vertical-beam count is $000 (start of field) and the long-field flip-flop (long field equals the even field of NTSC and PAL), which appears as bit 15 in the VPOSR/VPOSW registers, is set. The vertical-beam counter then counts out a long field. A short field follows before the next *VSYNC reset is needed, which indicates that another long field should start. This sequence implies that the genlock device should assert the *VSYNC signal every two fields at the beginning of the long field, which forces the Amiga to synchronize to the genlocks vertical position. Note that this discussion involves the custum chips in interlaced display modes only, because the NTSC and PAL composite video signals are by definition interlaced. Also note that the vertical-beam counter resets the same way in NTSC and PAL modes.

## *HSYNC AND *VSYNC RESET PULSE TIMINGS

With an understanding of the effects of a genlock's *HSYNC and *VSYNC reset signals on the Amiga's custom chips, we are ready to discuss the specific timing of applying these signals. In NTSC display modes, a genlock must provide the Amiga with horizontal and vertical reset pulses with the following characteristics:

- *HSYNC signal input: Active low pulse ≥ 2 color clocks (0.558µs) but ≤ 1 half-line duration (31.77µs), every 2 lines (127.11µs).

- *VSYNC signal input: Active low pulse ≥ 1 half-line duration (31.77µs) but ≤ 1 line duration (63.556µs), every 2 fields (33.36ms). Note that the *VSYNC reset pulse forces the next field to be a long (even) field.

In PAL display modes, the genlock again must provide the Amiga with horizontal and vertical reset pulses such that:

- *HSYNC signal input: Active low pulse ≥ 2 color clocks (0.563µs) but ≤ 1 half-line duration (32µs), every line (64µs).

- *VSYNC signal input: Active low pulse ≥ 1 half-line duration (32µs) but ≤ 1 line duration (64µs), every two fields

(40ms). Note that the *VSYNC reset pulse forces the next field to be a long (even) field.

A genlock must generate both reset pulses (*HSYNC and *VSYNC) at all times, in both NTSC and PAL display modes, regardless of whether or not the external source video is being input to the genlock! Refer back to Figures 1 and 2 for more details on the *HSYNC/*VSYNC reset signals.

### XCLK GENERATION

In addition to providing the proper *HSYNC and *VSYNC reset signals, a genlock must also supply a master dot clock (XCLK) that is synced with external video input to the genlock or, if no external video is input, that is self-synchronized at the correct frequency. When in self-synced mode, the clock should be stable and be crystal-generated.

One method of generating the master dot clock is to use PLL frequency multiplication in a PLL loop. The phase detector in the PLL block compares a 28.X MHz clock (XCLK signal) divided down to the line duration with the video source's line duration. This forms the basis for a line-locked clock generator. The phase detector's error signal is then input into the PLL's VCO, which generates the 28.X MHz dot clock. Exciting the VCO is a 28.X MHz crystal, which is slightly detuned so that it can follow or track the mean value in the incoming video's line duration. This may be necessary with VCRs because of their usually unstable output signals.

When in self-synchronized mode (no external source video), the aforementioned comparison can instead be made between the computer's *C1 (CCK) clock and the dot clock divided down by eight (instead of the line duration), and then input into the PLL's phase detector. Again, the error signal from the phase detector is input into the PLL's VCO, which is generating the 28.X MHz dot clock. Whatever the means you use to generate the master dot clock signal (XCLK), you must ensure that if no video source is present the dot clock free-runs at the correct 28.X MHz frequency (28.63636 MHz for NTSC, 28.375156 MHz for PAL) and remains stable with very little clock jitter and a good 50-percent duty cycle. For reference, there are 1820 clock periods of 28.63636 MHz in the NTSC line mode and 1816 clock periods of 28.375156 MHz in the PAL line mode.

### OS INTERLACE BUG

Since the original release of the operating system, the graphics library has contained a minor bug that does not force the LACE bit (bit 2) in the BPLCON0 register of Agnus if a genlock is attached to the system. This has spawned assumptions about the vertical-blanking interrupt handler and the rest of the system that affects system timers and their values. The bug has also resulted in a mismatch between the *CSYNC signal Agnus outputs and the interlaced composite sync portion of the composite video signal input to the genlock. Because some genlocks may use this signal, the mismatch can cause problems in certain display modes.

With OS 2.04, Commodore provides a simple program called Lacer that automatically sets the LACE bit if, and only if, a genlock is installed. Normally, the graphics library sets this bit if the user selects an interlaced display mode. If the user selects a noninterlaced display mode, graphics.library does not set the bit, which is the correct procedure. When a genlock is attached to the system, however, the genlock forces the output to interlace, because the NTSC and PAL video standards are interlaced. The problem is that in noninterlaced display mode, the LACE bit is still not set (affecting how the *CSYNC signal is generated) even though the genlock now forces the Amiga's output to interlace. As a result, the Amiga's *CSYNC signal doesn't match the NTSC or PAL standards, which may cause problems for some genlock designs. For example, if the system is displaying a noninterlaced image or is in noninterlaced display mode (200/256 lines) with a genlock attached, the output will interlace because the genlock forces the *VSYNC and *HSYNC input signals to reset the beam counters in an interlaced fashion.

Technically, the LACE bit should be set as the output is in fact interlaced. Therefore, the LACE bit can be set and still allow the system to run in 200/256-line modes; the image will interlace because the genlock is controlling the horizontal and vertical timing of the system. This is not a problem so long as the user understands that once a genlock is attached to the system, the Amiga's output operates at only NTSC or PAL display rates, which are always interlaced.

Setting the LACE bit in Agnus and assuming that the genlock is following the *HSYNC/*VSYNC reset timing shown in Figures 1 and 2 will, from an Agnus point-of-view, allow the *CSYNC output signal to match the interlaced composite sync portion of the composite video being input to the genlock. This should be true in whatever legal display mode the user selects with the ScreenMode preferences. To activate the "Lacer" program, the user simply double clicks its icon or, for automatic operation, drags the icon into the wbstartup drawer.

### GENLOCK DESIGN POINTERS

No amount of theory is useful if it is not well applied: Good design practices are the major part of any high-quality NTSC or PAL genlock design. And while ensuring that your genlock design conforms to the above interfacing details, it is equally important to conform as closely as possible to either the NTSC/RS-170A or PAL CCIR EBU video standards. (For a good reference on the NTSC/RS-170A standard, see "The NTSC/RS-170A Video Standard," p. 30, June/July '91.)

Pay attention to noise and color-burst ringing on the genlock's output as this can cause excessive chroma crawl on the user's display. Video levels are another area where close attention to the details of the video standards' specifications is very important. Follow the fine details of the specifications for both NTSC and PAL in the areas of SC-H phasing, timing tolerances, IRE tolerances, and rise/fall times on both burst and sync. In your design, attempt to have less than 1 percent differential-gain error and 1 degree differential-phase error (use high-quality test equipment to measure this) with a flat video bandwidth response up to at least 5 MHz. Make sure that your genlock's encoder is correctly and accurately doing its' job by checking the output on a vectorscope to be sure that with a color-bar test pattern the genlock's color vectors are on (or very close to) the small target boxes.

Finally, use high-quality video test equipment both when designing the genlock and calibrating it during manufacturing. Remember: Having the best genlock design in the world does you no good if the genlock is not calibrated correctly at the factory! ■

*Scott Hood is a Video Products Design Engineer with Great Valley Products. He formerly worked for Commodore on the A3000 and the A2320 Display Enhancer. Contact him c/o* The AmigaWorld Tech Journal, *80 Elm St. Peterborough, NH 03458.*

# A Tight Fit: JPEG Compression

*80-to-1 image compression? JPEG can give it to you, and more if you need it.*

By Perry Kivolowitz

THE PROVERB "A picture is worth a thousand words" must refer to low-resolution pictures, because high-resolution pictures often consume millions of WORDs (and bytes). JPEG image compression technology helps you tame out-of-control image size. As you will see, however, its compression is not without cost.

True-color images require eight bits (one byte) to represent their 256 levels of red, green, and blue information. Because three colors must be saved, each dot in the image requires three bytes (or 24 bits) of storage. This space requirement adds up fast. A video-sized image (768×480) consumes 1,105,920 bytes of storage, and a typical slide-sized image (2048×1356) fills 8,331,264 bytes. More staggering still are the sizes of many images from color scanners. An 8.5×11-inch page scanned at 300 DPI in color requires 25,245,000 bytes of storage.

Clearly, you need some kind of compression to handle these behemoths. The IFF format uses Run Length Encoding (RLE) compression. RLE compression simply identifies consecutive data that are all the same. Such "runs" are replaced with a single value and an indication of the number of times it should be replicated (the "run length"). Using RLE, you can expect an average compression ratio (size of the original uncompressed data compared to the compressed size) of about 2 to 1 or 3 to 1. As you might guess, the results depend heavily on the image you are compressing. An image dominated by a low frequency content (a fancy way of saying the image doesn't change much), such as one that's solid black, will compress very well. An image dominated by a high frequency content (it changes a great deal), such as television static, won't compress well at all.

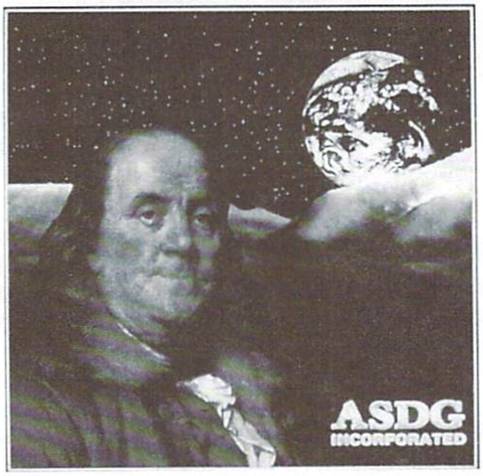


Figure 1: The original 24- bitplane image.

On the average, video-sized images compress to about 400,000 bytes, for a ratio between 3 to 1 and 2 to 1. Slide-sized images compress to about 3,100,000 bytes, and so on. While RLE-compressed images are smaller than uncompressed images, they're still pretty bulky. This is the major shortcoming of RLE compression. Its advantages are that it executes pretty fast, and that the image you get back (after compressing and decompressing) is exactly the same as the image you started with.

The technical term for "getting back the image you started with" is "lossless" image compression—lossless, because no picture information is lost during the compression and decompression steps. TIFF and GIF file formats use the LZW lossless image-compression technique, which can give compression ratios as high as 6 to 1 on the average. Even at 6 to 1, however, a slide-sized image would still consume 1,388,544 bytes of storage. To really tame the image-size problem, we need even heavier compression than this.

To see why image compression is so important, think of it this way. On your 80-megabyte hard drive you could store only ten slide-sized images in their uncompressed form—even less, if you have to store operating-system files and program files on the same disk. At 6-to-1 compression, your hard drive could store an average of 60 slide-sized images. But, what if you could achieve 80-to-1 compression? Then you could store more than 800 slide-sized images!

## JPEG COMPRESSION

The JPEG image-compression method can achieve ratios as high as 80 to 1 or even higher. The name stands for the Joint Photographic Experts Group, the ISO committee that created the compression method. JPEG is a "lossy" technique: To achieve the incredible results it is capable of, JPEG may modify your data so that the image you get back (after compressing and decompressing) will not be the same as the one you started with.

The picture information is not lost indescriminantly. Part of the JPEG algorithm is based upon how people perceive a visual stimulus. JPEG "knows" how to remove some details (which will greatly improve compression) that most people would never miss. Plus, you control how much loss takes place by telling the JPEG algorithm how much compression it should perform. Therefore, you decide what is more important to your application on an image-by-image basis. Is image quality (closeness to the original image) important? Or is space savings? You are in command.

Encoding an image using JPEG compression usually begins with a color space conversion from RGB to YUV (also

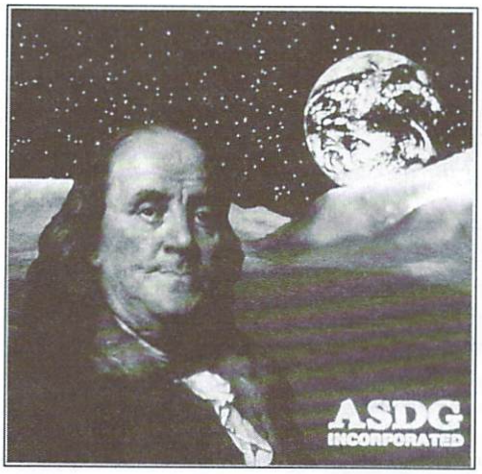



Figure 2: The luminance (Y) component of the image.

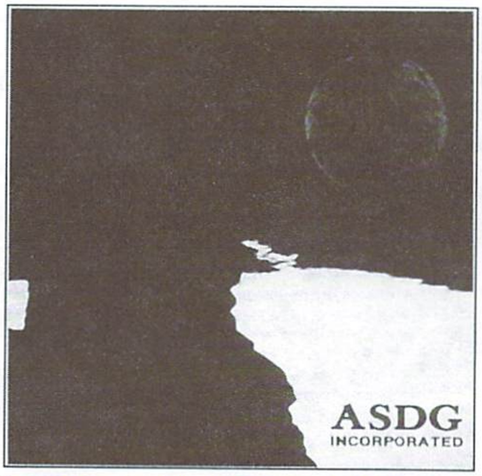



Figure 3: The U or chrominance-blue (Cb) component of the image, contrast enhanced.

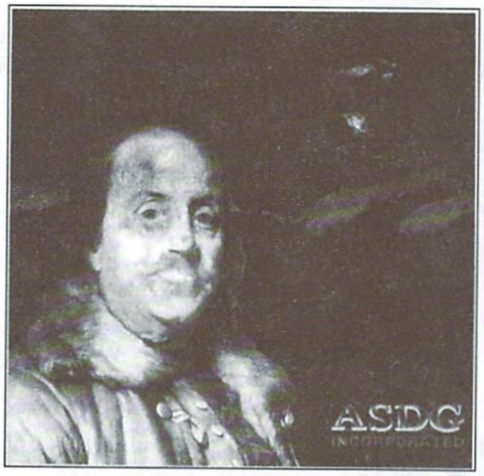



Figure 4: The V or chrominance-red (Cr) component of the image, contrast enhanced.

called YCbCr). Technically, the JPEG specification does not mandate this step, but doing so can improve compression performance. The reason is based on the derivation of the YUV color model. The designers of the PAL and SECAM video standards found (as did the designers of the NTSC video standard) that the information in a picture can be split into three classes (in much the same way as splitting an image into red, green, and blue "classes"). The human visual system turns out to be very sensitive to one of these classes (the Y component), and less so to the other two (U and V).

To show this, let's introduce Figure 1, which I'll use as the benchmark. The image in Figure 1 is 960×960 pixels in size, requiring 2,764,800 bytes uncompressed (the original was in color). IFF, using RLE compression, can trim the space requirement to 1,462,298 bytes (almost a 2-to-1 compression).

Figure 2 shows the luminance, or Y, component of the original image. Notice how this image resembles what you would see if the original color image were displayed on a black-and-white television. This similarity is more than passing; the luma component of a color television broadcast is what is displayed on a black-and-white television.

Figure 3 shows the U component of the image. The U component is also called the Cb component, standing for "Chrominance blue." This is because the U component heavily penalizes both the red and green information in the image, leaving mostly blue information. Similarly, Figure 4 shows the V, or Cr ("Chrominance red") component of the image. Note how it has detail only in the areas of the picture that contained a lot of red (the jacket, lips, and cheeks).

Both Figures 3 and 4 have been enhanced with Art Department Professional to show detail. In fact, the unprocessed versions of these images are mostly black and have very little detail. With this in mind, comparing these images to Figure 2 shows you just how luminance-oriented the human visual system is. That is, having detailed information in the luminance portion but skimpy information in the chrominance portion still produces a correctly appearing image.

This fact accounts for one of the ways that JPEG compression considers the way people see in choosing which details to omit. A JPEG compressor using YUV input will compress the Y information less than it does the U or V components, knowing that people will notice changes to U and V less than changes to Y.

The next step in the JPEG compression procedure is to transform small chunks of the Y, U, and V portions of your image using a technique called the Discrete Cosine Transform (DCT). Specifically, the image is compressed in 8×8-pixel blocks (64 pixels at a time). This size was chosen for several reasons, including the fact that an 8×8 area is reasonably large yet is likely to contain pixels of similar color (an 8×8-pixel area is likely to be dominated by a low frequency component).

DCT is a cousin of the Fourier transform. It takes the information in the 64 source pixels and produces an array of 64 coefficients that represent the "energy" contained in the source pixels. A critical feature of the coefficients array (and what makes the DCT so useful) is that the first coefficient represents the most "energy" and the amount of "energy" represented by the remaining coefficients drops off very quick- ►

ly. In other words, most of the information contained in the original 8×8-pixel array is represented by the first element of the DCT transformed array. This step introduces a bit of the loss associated with JPEG in that the DCT is not perfectly invertible (during the decompression step). This contribution to the loss, however, is minor compared to the next step.

The transformed array of 64 pixels is then quantized to reduce the amount of variance in the information it contains. This is where the lossy nature of JPEG really comes in. When an image is compressed heavily, much of the variance in the coefficient array is eliminated. Therefore, during decompression a different set of coefficients will be turned back into picture information by an inverse DCT. The quantization step ensures that the resulting data (which no longer varies much) will compress well with a standard lossless data compression technique.

Because the U and V components can be quantized more dramatically than the Y component, the U and V components will compress more heavily in the last step of the JPEG compression process. As a side effect, there may be more loss in the U and V components, because the coefficients generated during the decompression process are more likely to differ from the original coefficients during the compression process.

As mentioned, the last step of the compression process is to use a standard lossless compression technique to encode the quantized DCT coefficients. The technique most widely used for this phase is a one-pass Huffman encoding. In this technique, the bits comprising the quantized coefficients for the whole image are examined in succession. As patterns are found to repeat, they are assigned shorter "tokens." The relative size difference between the tokens and the bit strings they represent determine the amount of com-


Figure 5: The color original of the image at left was compressed almost 16 times. At right, the differential between the compressed and the original image.


Figure 6: The color original of the image at left was compressed 44 times. At right, the differential between the compressed and the original image.


Figure 7: The color original of the image at left was compressedalmost 118 times. At right, the differential between the compressed and the original image.

pression that takes place.

Decompression takes the above steps in reverse order. First, bit strings are decompressed into quantized DCT coefficients. An inverse DCT is performed, turning these coefficients back into picture data. The color space of the picture data can then be converted back into its original form. Again, the loss associated with JPEG crops up during the DCT and inverse DCT steps to a small degree, and during the quantization step to a larger degree.

One enhancement to the JPEG algorithm that many JPEG implementations have (including ASDG's JPEG implementation for Art Department Professional) is a smoothing step performed during decompression. Recall that JPEG compresses your image in 8×8-pixel chunks. Often, the loss that occurs during compression is not as noticeable as the discontinuities that can crop up at the borders of each 8×8-pixel area. Smoothing during decompression can help hide the borders between compression chunks. The smoothing operation itself can be performed with a knowledge of where the 8×8-pixel borders are located, thus only a fraction of the image is smoothed.

### JPEG EXAMPLES

The three pairs of imagesin Figures 5–7 demonstrate successive levels of compression. Note well that the smoothing step described above was *not* employed in these examples so that you could see exactly what the baseline JPEG algorithm does at successively higher levels of compression.

The first image in each pair shows the image after being compressed and then decompressed. Compare these images with the original image shown in Figure 1. The second image in each pair is a differential image provided to help you locate the differences between the compressed/decompressed images and the original. These images were created with Art Department Professional by subtracting the JPEG processed image from the original unprocessed image. The resulting images are shifted so that where there is no difference between the original and processed images, a neutral grey shows in the printed image.

The ideal result would be a perfect solid grey (something minus itself is nothing, shifted up to neutral grey). By examining the differential images for deviations from neutral grey, you will be able to spot the changes (or loss) caused by JPEG compression.

Figure 5 shows the result of the ASDG JPEG implementation when a "quality level" of 50 was used. In ASDG's implementation, you can enter a quality level to control the amount of compression that will take place. The quality level ranges from 0 to 100, where the higher the number, the higher the quality (and less compression). Different JPEG implementations provide different levels of compression control, so our quality level 50 need not correspond to another JPEG implementation's quality level 50.

At quality level 50, the original 2,764,800-byte image was compressed to just 174,493 bytes, yielding a compression ratio of 15.85 to 1. Notice that the differential image in Figure 5 is almost a perfect solid grey. This is proof that your eye is not deceiving you when you decide you can't really see any difference between the JPEG-processed image in Figure 5 and the original image. In ASDG's implementation, the default quality level is 32. This means that the image in Figure 5 would have been compressed more heavily using the default settings.

Figure 6 shows the result of JPEG compression using quality level 10. At this level, the original image was compressed to just 62,255 bytes. This yields a compression ratio of 44.4 to 1. The differential image in Figure 6 should show you where there are changes caused by JPEG. As you can see, however, at this compression level the results are still good, even though the image is compressed more heavily than with the default settings.

Finally, Figure 7 shows the result of JPEG compression using quality level 1. Here, the original image was compressed to just 23,427 bytes, for a compression ratio of 118 to 1. At this compression level you can readily see the differences between the compressed image and the original. Clearly, for this particular image, we've compressed too far.

### JPEG TIPS AND PITFALLS

Having JPEG on the Amiga is a very important step forward for our machine, because it makes managing larger and larger collections of high-resolution color images possible. Without a compression technology, such as JPEG, those using the Amiga for professional imaging applications would find themselves at a disadvantage; they'd have to buy either huge hard drives or make do with limited image libraries. Also, the small size of JPEG compressed images makes transmitting images via modem a much more reasonable proposition. As a result of JPEG availability, I predict that Amiga users will begin sharing 24-bit true-color images in much the same way that HAM and 16-color images are shared today.

Clearly, JPEG usage must be tempered with some knowledge of its limitations. For example, knowing how you will use an image helps you decide how much compression you want (how much loss you are willing to put up with). Also, the content of each image being compressed plays a role in how far you can compress it. A picture of a human face compresses much better than a picture of text.

As a bonus, you can use JPEG files created with the ASDG implementation on other computers that also have JPEG support complying with the JFIF convention. JFIF, the JPEG File Interchange Format, is the standard for storing JPEG-compressed files. Such a standard is necessary because JPEG defines only the compression method, not the file format. To understand the difference consider this example: Both IFF and PCX use RLE compression. Although the two file formats use the same compression method, an IFF reader can't understand a PCX file or vice versa.

I will close this article with a description of another way to measure compression. You are already familiar with the concept of a compression ratio—the relative size of the original image as compared to the size of the compressed image. Most people don't really feel the full impact of things expressed as ratios, however, because ratios are fuzzy mathematical concepts. So, consider that a true-color pixel is made of three eight-bit quantities, for a total of 24 bits. If you compress an image 40 to 1, then each 24-bit pixel is represented in a JPEG file by just *six-tenths of one bit*! Each of the 24-bit pixels in Figure 7 was represented in the JPEG file by just *two-tenths of one bit*. In other words, each *bit* in the JPEG file for Figure 7 held the equivalent of five 24-bit pixels. JPEG *is* incredible. ■

*Perry Kivolowitz is cofounder of ASDG Inc., which specializes in color image processing. Contact him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458 or on BIX (perry).*

ON DISK

# Blitter Optimization

*With interleaving you can blit your images up to twice as fast!*

By Leo L. Schwab

THE AMIGA IS great for games: Not only does it have the horsepower to do visually stunning things, but the rich set of tools in the graphics library makes getting your game up and running very easy.

One of the most frequently performed operations in a game is blitting an image onto the screen. Thus, if you have a game that has lots of blitted objects, you want the blits to happen as quickly as possible. Unfortunately, the blit routines in graphics.library, because of their generality, aren't up to the demands of a high-performance game. This frequently leaves you to write your own blit routines, which can be a daunting task, even with Tom Rokicki's excellent articles on the subject (such as "The Complete Guide for the Blittering Idiot," p. 2, October '91). Help is here: By using the OS-routine-based blitting method outlined below, you can increase blit performance by as much as a factor of two.

Let's examine the problem first. The Amiga uses bitplanes to represent imagery in memory. To blit an image, each plane must be blitted separately. For each plane, this involves fetching plane pointers, waiting for the Blitter to become idle, performing Blitter setup, and starting the blit.

The Blitter itself, once running, takes a variable amount of time to complete its work; this time is more or less directly proportional to the size of the area being blitted. Blitter set-up time, however, remains a constant value regardless of blit size. Moreover, set-up is done for each plane, and is multiplied by the number of planes that need to be blitted. For a 32-color image, the time can become significant enough to affect performance. Because the Blitter itself is a piece of hardware whose speed we can't control, our only recourse is to attempt to reduce the set-up time for each full blit.

The dreary scenario above assumes that each plane must be blitted separately. Of course, if we were blitting just one bitplane, the Blitter setup would occur only once, and the performance hit would be minimal. Hmm. Wouldn't it be nice if there were a way to blit multiple bitplanes with a single pass of the Blitter?

There is. By forcing a special arrangement of bitplanes in memory, you can fool the OS into thinking that any number of bitplanes is really just one. Thus, only one blit is necessary, and the set-up time for the additional planes vanishes. (Poof!)

### TILT YOUR BRAIN SLIGHTLY TO THE LEFT

This method will require you to shift your thinking about bitplanes a little. For this discussion, the width of each bitplane in pixels will be represented by "WIDE," the height by "HIGH," and the total number of planes by "DEEP."

Ordinarily, bitplanes are scattered throughout chip memory, as in Figure 1. This is the least constraining method of bitplane arrangement and is the model that the OS uses. This method, however, requires separate blits for each plane and is not efficient. Our approach allocates bitplanes in memory as a single block. The size of the block is equal to the size of one bitplane times the total number of planes.

That's the first step. The second step is deciding how to represent this block of memory to the OS blit routines. At first, you might simply divide the block into N equal pieces, one for each plane, as suggested by Figure 2. This doesn't buy us

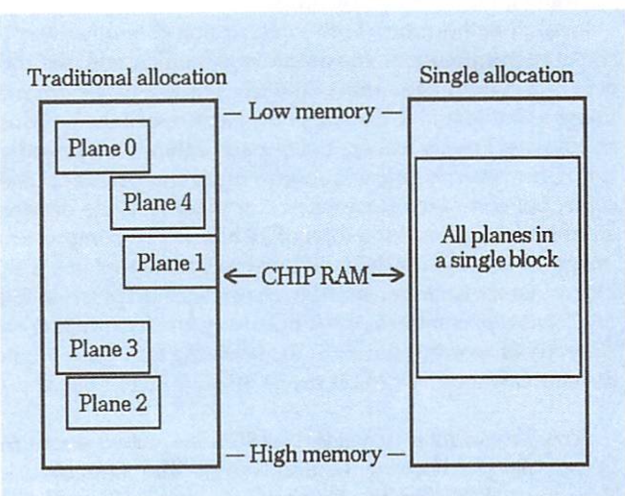



Figure 1. Bitplane allocation methods.

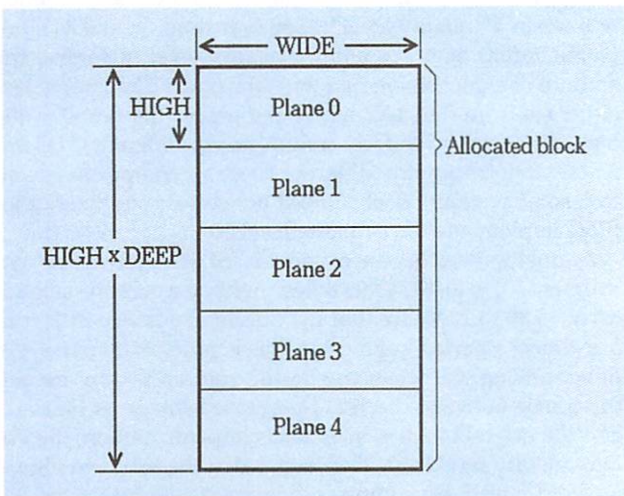



Figure 2. How not to partition the block.

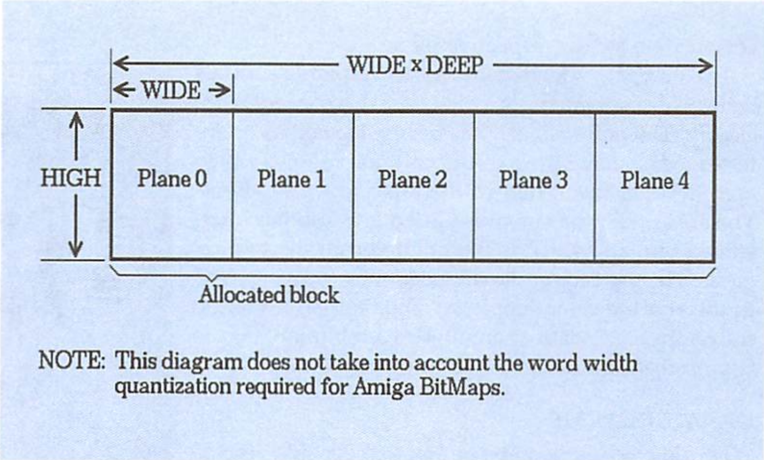



Figure 3. Proper partitioning strategy.

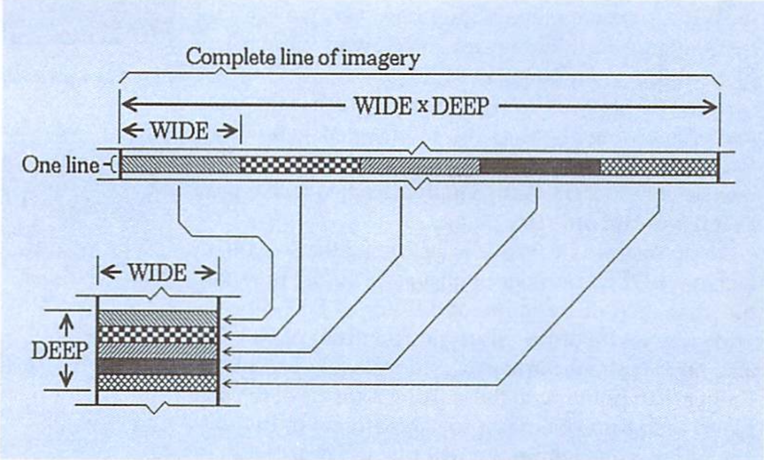



Figure 4. Interleaving transformation.

anything, though, because it's substantially the same as allocating separate bitplanes that just happened to wind up next to each other in memory. We need to merge the planes so that they can appear as a single unit.

Figure 3 shows part of the secret. We can represent the planes as being stacked "sideways." That is, the bytes for line 0, plane 0 are immediately followed by the bytes for line 0, plane 1, followed by the bytes for line 0, plane 2, and so on. After reaching the end of the bytes for line 0, plane N, we find the bytes for line 1, plane 0, followed by line 1, plane 1, and so on.

The advantage of this geometric configuration is that a complete line of imagery is stored in a continuous region of memory. Thus, to manipulate one complete line, we have one big operation, rather than several small operations scattered through memory. More generally, the whole bitmap can be treated as a single plane, which is roughly WIDE times DEEP in width, and HIGH lines tall.

What I described above is what I call the "short-and-fat geometry," a convenience that permits us to still think in terms of bitplanes. There is one more geometric configuration to consider, however, which is the configuration that will be represented to the OS blit routines.

Let's take the short-and-fat geometry and shuffle it. Consider a single complete line of imagery. In the previous arrangement, each complete line was composed of its planar components, arranged horizontally. Now let's take the planar components and arrange them vertically. That is, instead of stacking the planar components of each line end to end, we stack them on top of each other. This transformation is called "interleaving," and is illustrated in Figure 4.

What we're left with is what I call the "tall-and-thin geometry." Like the short-and-fat geometry, the entire image can be treated as a single plane, which is WIDE pixels in width, and is HIGH times DEEP lines tall.

Note that the actual arrangement of the bytes in memory is precisely the same between the two geometries; only the outward representation of the bits is changed. It's rather like taking a two-dimensional array you set up with dimensions [2][5] and setting up a new pointer that pretends it's [5][2]. The total size of the array hasn't changed, nor have any of the actual values in it; only the way you index them has changed.

Great. So what has all this brain-bending bought us, other than a unique new form of headache?

Let's assume that both the source and destination bitmaps are of the same geometry. Recall that, in the interleaved (tall-and-thin) geometry, every consecutive group of DEEP lines in the interleaved plane represents a complete line of imagery. Thus, to blit a complete line, we merely specify the width we want in the usual manner, and specify a height of DEEP. The blitter will then go off and copy DEEP lines from the source to the destination, thereby copying a complete line of imagery. More generally, to copy an arbitrary number of complete lines, we multiply the desired height by DEEP. The Blitter will then do the operation, copying each complete line until done.

Poof! We just manipulated a complete image in a single blit! This concept can be difficult, so let's step through it.

We allocate a single bitplane whose width is (roughly) WIDE times DEEP, and whose height is HIGH. We divide this short-and-fat plane into DEEP separate horizontally adjacent virtual regions. Each region represents a single plane in the final image. In each "plane" we place the individual bitplanes of the image.

Next, we interleave the lines in the bitplane, yielding a tall-and-thin bitplane. Recall again that the actual bits in RAM are not touched; this transformation is performed by merely altering the values in the bitmap structure.

We then blit the interleaved bitplane to another bitplane that has also been interleaved. This blits the complete image.

Now, "unwind" the destination bitplane back into a short-and-fat bitplane. The bits in RAM are not affected. You will find that each virtual region in the destination contains the ►

correct data for the respective plane.

The program ViewBlit in the accompanying disk's Schwab drawer attempts to illustrate this concept more clearly. The entire screen has been allocated as an interleaved custom bitmap. The box cursor shows which area is being illustrated in the top and side regions. The area under the cursor is blitted into another interleaved bitmap, which is then blitted into the two regions. The top region shows the bitmap when viewed in the short-and-fat geometry. The region on the left shows the same data as an interleaved bitmap. Figure 5 is a screen capture of the running program.

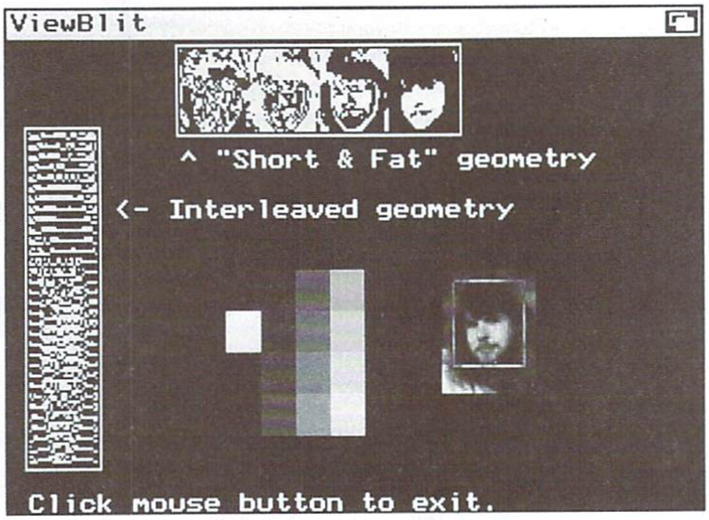



Figure 5. An output screen from ViewBlit.

## OKAY, SHOW ME

For this technique to be of any real use, we need to be able to get the interleaved image onto the screen. The Amiga can't display an interleaved bitmap directly. With some creative fibbing, however, we can fool the Amiga into thinking our interleaved bitmap is a planar bitmap, and display it.

Recall the short-and-fat geometry (which is the same memory arrangement as the interleaved geometry). Each "plane" in that orientation is a complete unit. We need to convince the Amiga to treat each virtual region as if it were an ordinary plane.

To do this, we lie. We tell the Amiga that we do, in fact, have DEEP number of bitplanes, HIGH lines tall, but that each of them are WIDE times DEEP pixels wide (the width of the entire short-and-fat plane). We also lie to it about the starting address for each plane. Rather than pointing to the starting address of the real plane each time, we point to successively more horizontal positions within the real plane. These horizontal positions just happen to lie at the boundaries of the virtual regions in the short-and-fat plane. The effect of this is illustrated in Figure 6.

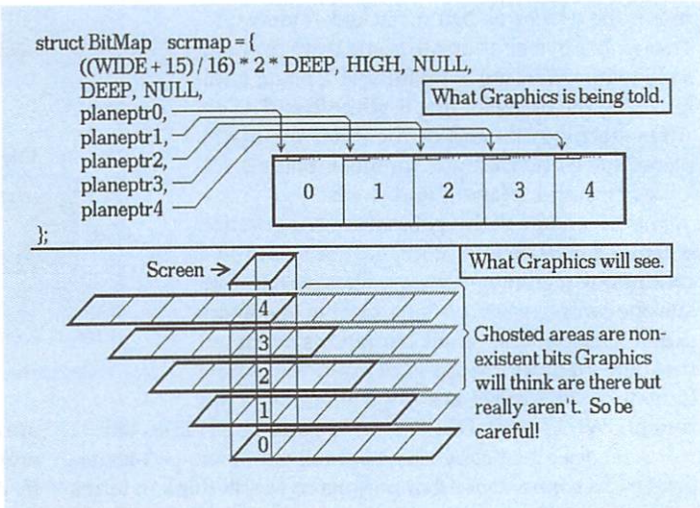



Figure 6. Constructing an interleaved display.

We can then take this fabricated bitmap and pass it to OpenScreen() as a custom bitmap. Intuition then happily composes a display based on this bitmap. As far as Intuition is concerned, this is an ordinary screen. You can put windows, menus, and requesters on it, and perform traditional blits on it. Plus, you can perform interleaved blits as well.

Keep in mind that, when you do this, you are effectively telling graphics that you have planes that extend farther to the right than they really do. This means that if your rendering strays outside the limits of WIDE and HIGH, you will draw pixels into The Bizarro Zone, and possibly bring the machine down. In particular, SetRast() is unsafe, because it will try to affect the entire bitmap. Be extremely careful when rendering through the fabricated bitmap. If you anticipate problems, open a window on the screen and let layers do the clipping for you.

The actual code to make all this happen is very simple. To allocate an interleaved bitmap, you do the following:

```
// Format bitmap to one plane, width WIDE, height HIGH * DEEP
InitBitMap (bm, 1, WIDE, HIGH * DEEP);
if (!(bm.Planes[0] = AllocRaster (WIDE, HIGH * DEEP)))
fail ("Can't allocate bitmap.\n");
```

Note that we multiplied the height by the depth. To have instead multiplied width by depth would have been insufficient because of the need to round up to the nearest multiple of 16 pixels (the width of a WORD) per plane. For example, if WIDE were 40, the required BytesPerRow for each plane would be six. For a depth of five, 40 * 5 is 200, which yields a BytesPerRow of 26. However, the correct value is 30 (six BytesPerRow for each plane, times five planes). Multiplying the height by the depth avoids this pitfall.

To create a short-and-fat bitmap based on an interleaved one, you use:

```
InitBitMap (sfbm, 1, WIDE, HIGH);
sfbm->BytesPerRow *= DEEP;
sfbm->Planes[0] = bm->Planes[0]; // 'bm' is interleaved.
```

Note the multiplication of BytesPerRow by the depth. Again, this is to preserve the word width quantization required for Amiga bitmaps.

To create a fabricated bitmap suitable for display, you use:

```
InitBitMap (displaymap, DEEP, WIDE, HIGH);

// Widen BitMap to generate correct display modulos.
displaymap->BytesPerRow *= DEEP;
```

```
for (i = 0; i < DEEP; i++)
// Start planes at successively more horizontal
// positions, which just happen to lie on the
// boundaries of the "virtual regions."
// (bm points to our interleaved bitmap.)
displaymap->Planes[i] = bm->Planes[0] + bm->BytesPerRow * i;
```

So after we've gone to all this trouble, does it actually buy us anything? Do blits really happen faster?

The example program OptiBlit in the Schwab drawer illustrates the speedup. The program performs 10,000 blits of a 32×32×5-plane image into an Intuition screen, first using traditional blits, then using interleaved blits. The times consumed by these operations are recorded and printed.

The table below shows the results:

| | Times (secs) | | % Time Saved |
|---|---|---|---|
| | Normal | Interleaved | |
| 1.3 68030 FastROM | 14.29 | 10.55 | 26% |
| 1.3 68030 | 20.62 | 11.34 | 45% |
| 1.3 68000 | 22.23 | 10.39 | 53% |
| 2.04 68030 FastROM | 12.25 | 10.78 | 12% |
| 2.04 68030 | 12.59 | 11.56 | 8% |
| 2.04 68000 | 15.04 | 11.91 | 21% |

**System configurations:**

**68030: Amiga 2500/30, 1MB CHIP, 2MB FAST (32-bit).**

**68000: Amiga 2500/30 (68000 mode), 1MB CHIP.**

**Times shown were collected by running OptiBlit five times under each configuration and averaging the results. 2.04 tests were conducted with Workbench running, using a production ROM.**

On an A2500/30 running 1.3, using SetCPU FASTROM, interleaved blits take 26% less time than traditional blits. Without FASTROM, the figures become even more startling. The biggest gains are found on a 68000-based system. Under Kickstart 2.04, the blit functions in the graphics library underwent heavy optimization, so the results are less dramatic.

## GRAVY

Faster blits are not the only benefit of interleaved bitmaps. One additional benefit is simpler loading of IFF ILBM images. ILBM, in fact, stands for InterLeaved BitMap. Ordinarily, you need to distribute the BODY data into the separate planes when loading and update all the respective plane pointers. If you load the BODY into a bitmap that is already interleaved, however, no separation is necessary, and you need only keep track of one pointer.

Another significant benefit is the predictability of blits. Using traditional blits on a display some "hashing" usually results, which is caused by the video beam showing parts of the image that haven't had all planes updated yet. With interleaved blits, complete lines are processed in order. Recall that the planar components of a line of imagery are stored consecutively. Thus, the Blitter progresses down the screen, rendering complete lines one after another. Hashing is still possible, but the visual effect will be some fraction of a complete image, with perhaps a single garbled line. This is far less jarring than the hashing seen using traditional blits. Also, because Blitter rendering is more integrated, programming of beam-avoidance techniques becomes much easier.

If, as mentioned earlier, you have been custom programming the Blitter (in an effort to reduce set-up times), you can still use this technique to your advantage. In fact, the combination of your reduced set-up time with the ability to do just single blit should make your application scream.

## TANSTAAFL

Of course, interleaved blitting does have its set of drawbacks. Chief among these is the need for larger regions of contiguous chip RAM, because bitplanes must be kept together. As an example, if you want to do interleaved blitting on a 640×400, 16-color screen, you will need to be able to allocate all four planes in a single chunk, which is 128,000 bytes. On a plain Amiga 500 with 512K of chip memory, it's unlikely you'll get a chunk that size unless the system has been freshly booted or happens to be relatively unfragmented.

If you are doing blits through masks (cookie-cut blits), the size of your masks could increase substantially. With traditional blits, each plane is blitted through a single-plane mask. With interleaved blits, the same is true. However, the mask must match the size of the single interleaved plane, which is DEEP times taller than the traditional form. Thus, the size of your masks could become prohibitive. This problem could be alleviated by using traditional blits when rendering the masked shape into the display, and using interleaved blits when restoring the background.

Using this approach, it is quite easy to run into the maximum size of a single blit, which is 1008 wide by 1024 high. (Note: Kickstart 2.04 with the ECS chips increases this limit to 32K×32K.) For example, when interleaved, a 320×400, eight-color screen yields a bitplane 1200 lines tall, which is more than the blitter can handle in one pass. Clearing such a screen could require more than one blit. (If you've followed this article, you can clear this bitplane in one blit. I leave this as an exercise.) However, this is an issue *only* when attempting a blit of such size. It doesn't matter how big the source or destination bitmaps are. As long as the total size of the blit is less than 1008 wide and 1024 high, it will work just fine.

If you create a screen using an interleaved bitmap, some screen-grabbing programs may be confused by the unusual geometry. Others may attempt to grab all the bits represented in the fabricated bitmap, yielding a much wider ILBM file than you expected.

It isn't possible to take advantage of interleaving when using the Blitter to draw lines into the bitmap. Also, drawing a solid rectangle of arbitrary color is not easily done (the only easy ones are all zeros and all ones). These must still be done using traditional bitmaps.

Note also that merely creating an interleaved screen doesn't mean that everything that happens in it will be faster. Intuition will still treat it as a planar display, and all its rendering will be at standard speeds. The only interleaved blits that will happen are those you do yourself.

Interleaved blitting is a common technique used by better game programmers. For some reason, however, nobody talks about it, leaving everyone to discover it for themselves (like I did). I have found this trick to be simple and effective, the drawbacks manageable, and the benefits well worth the effort. If you're planning on writing a game, or any other application that would benefit from faster blits, consider using this method to increase performance. ■

*Leo L. Schwab is the principal programmer of Disney Presents...The Animation Studio. He also worked on the CDTV ROMs. Contact him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (ewhac), Portal (ewhac), or usenet (ewhac@well.sf.ca.us).*

# Graphics.Library's Low-Level Drawing Routines

ON DISK

By Paul Miller

THE GRAPHICS LIBRARY is the Amiga's warehouse of low-level graphics rendering functions. These functions may be called primitives, but like the club of your typical caveman, they can be quite powerful when used properly. These primitives provide means for drawing lines, circles, polygons, fills, and text. They can also manipulate bitmaps and rasters and handle fonts, sprites, regions, scrolling, color palettes, and animation—not to mention control the Copper and Blitter. Whew! Because they have direct access to the graphics hardware, they are very fast—something some of us take for granted. Ask anyone who has been forced to use a Mac or Windows on a 386 PC.

It is no wonder that other Amiga libraries are built on the graphics library, including Layers and Intuition. Plus, if you like, you can use these primitives yourself, usually with little additional work. You can often maximize your control and efficiency by accessing the primitives in graphics.library directly, thereby minimizing the overhead normally incurred going through the other libraries.

With release 2.0 of the operating system, there are over 130 functions in graphics.library. (See "Graphics Handler," p. 12, November/December '91, for a complete description of 2.0's graphics.library.) Only a few of these, however, are used for low-level rendering. We're going to concentrate on the display system and drawing primitives.

### A CLEAN SLATE

Before you can start drawing with the graphics library, you'll need to set up the graphics display for the resolution and number of colors you want. This means reserving some chip memory for the bitmap and then telling the graphics hardware how to display this memory. You'll also want a way of telling the graphics primitives how to render into your memory, so you'll also need to create a RastPort for this bitmap. The RastPort keeps track of current drawing pens, drawing mode, write mask, line and fill patterns, pen position, font and font attributes, animation information, and other internal details.

There are two ways to set up the display: You can use the graphics library to create a very low-level display with no direct means for input, or you can let Intuition do it for you by opening a Screen. Both methods achieve the same goal and have drawbacks. If you use graphics.library, you will have to provide a way of getting input to the program, most likely by setting up your own input handler. On the other hand, Intuition does this for you, but it is a tad slower, unless you talk directly to the Screen's bitmap. We all know that easy input is "a good thing to have." However this article is, after all, on the graphics primitives, so I'll show you both ways. Once you get the display set up, the rest is similar (depending, once again, on how you plan to talk to your application).

For the example programs, let's set up a standard 320x200 screen with three bitplanes (eight colors).

Using the primitive method (as the Primitives program in the accompanying disk's Miller drawer does), you need to allocate a bitmap and its memory, set up a View, ViewPort, and RasInfo for displaying this memory, and hook a RastPort to it for drawing. Sounds easy, eh? Take a look:

```
#define WIDTH     320
#define HEIGHT    200
#define PLANES    3

struct GfxBase *GfxBase = NULL;

struct View view, *oldview;
struct ViewPort viewport;
struct BitMap bitmap = {0};
struct RasInfo rasinfo = {0};
struct RastPort rastport = {0};

/* notice I don't do any error checking here—you would normally want
to do it */
GfxBase = (struct GfxBase *)OpenLibrary("graphics.library",0L);

/* save a pointer to the current View to restore later */
oldview = GfxBase->ActiView;

InitView(&view);
InitVPort(&viewport);
view.ViewPort = &viewport;

viewport.RasInfo = &rasinfo;
viewport.DWidth = WIDTH;
viewport.DHeight = HEIGHT;
viewport.ColorMap = GetColorMap(1<<PLANES);

InitBitMap(&bitmap, PLANES, WIDTH, HEIGHT);
for (i = 0; i < PLANES; i++)
{
   bitmap.Planes[i] = (PLANEPTR)AllocRaster(WIDTH, HEIGHT);
   BltClear(bitmap.Planes[i], RASSIZE(WIDTH, HEIGHT), 0);
}

rasinfo.BitMap = &bitmap;
```

```
MakeVPort(&view, &viewport);
MrgCop(&view);
LoadView(&view);
```

Note that the only library you need to open is graphics. library. Also note that because the system dynamically allocates some internal structures, you'll need to explicitly instruct the system to free that memory when you're through:

```
if (oldview) LoadView(oldview);
WaitTOF();
for (i = 0; i < PLANES; i++)
{
   if (bitmap.Planes[i])
FreeRaster(bitmap.Planes[i], WIDTH, HEIGHT);
}
if (viewport.ColorMap) FreeColorMap(viewport.ColorMap);
FreeVPortCopLists(&viewport);
if (view.LOFCprList) FreeCprList(view.LOFCprList);
if (GfxBase) CloseLibrary((struct Library *)GfxBase);
```

If you prefer Intuition and its benefits (remember that you can't drag your display around when working with ViewPorts directly), your task is a bit simpler. In addition to graphics.library, you'll need to open intuition.library. The display information is sent to OpenScreen(), which does all the internal allocations for you. All you have to do now is set a pointer to the screen's or a window's RastPort and you can start drawing. A call to CloseScreen() later handles all of the deallocations for you. Because screens don't do clipping, this method will be about as fast as managing the display yourself. If you were rendering to a window's RastPort, there would be some additional overhead caused by clipping, which is still very efficient. The Area program (in the Miller drawer) manages its display with Intuition.

```
struct GfxBase *GfxBase = NULL;
struct IntuitionBase *IntuitionBase = NULL;

struct Screen *screen = NULL;
struct Window *window = NULL;

struct NewScreen ns;
struct NewWindow nw;

GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 0L);
IntuitionBase = (struct IntuitionBase *)
OpenLibrary("intuition.library", 0L);

memset(&ns, 0, sizeof(struct NewScreen));
ns.Width = WIDTH;
ns.Height = HEIGHT;
ns.Depth = PLANES;
ns.DetailPen = 0;
ns.BlockPen = 1;
ns.Type = CUSTOMSCREEN | SCREENQUIET;

screen = OpenScreen(&ns);
```

Under 2.0, matters are simpler:

```
UBYTE pen_array[] = {~0};    /* give it the "new look" */

screen = OpenScreenTags(NULL, SA_DisplayID, LORES_KEY,
                              SA_Depth, PLANES,
                              SA_DetailPen, 0,
                              SA_BlockPen, 1,
                              SA_Quiet, 1,
                              SA_Pens, pen_array,
                              TAG_DONE);
```

Now open a Window for input/clipping with:

```
memset(&nw, 0, sizeof(struct NewWindow));
nw.Width = WIDTH;
nw.Height = HEIGHT;
nw.DetailPen = -1;
nw.BlockPen = -1;
nw.IDCMPFlags = RAWKEY | MOUSEMOVE | MOUSEBUTTONS;
nw.Flags = BORDERLESS | BACKDROP | REPORTMOUSE | RMBTRAP
| ACTIVATE | SIMPLE_REFRESH;
nw.Type = CUSTOMSCREEN;
nw.Screen = screen;

window = OpenWindow(&nw);
```

Or under 2.0 with:

```
window = OpenWindowTags(NULL, WA_Width, WIDTH,
                              WA_Height, HEIGHT,
                              WA_IDCMP, IDCMP_RAWKEY
                                |IDCMP_MOUSEMOVE |
                                IDCMP_MOUSEBUTTONS,
                              WA_Borderless, 1,
                              WA_Backdrop, 1,
                              WA_ReportMouse, 1,
                              WA_SimpleRefresh, 1,
                              WA_Activate, 1,
                              WA_RMBTrap, 1,
                              WA_CustomScreen, screen,
                              TAG_DONE);
```

You can get access to the Screen's RastPort with:

```
rp = &screen->RastPort;
```

which allows you to render directly into the Screen's memory.

You obtain a pointer to the Window's RastPort via:

```
rp = window->RPort;
```

For internal use, some primitives (area functions and flood-fill) require a temporary single-plane raster that is at least as large as the RastPort being drawn into. Additionally, the area routines need some memory for holding vectors. The area information and temporary raster are linked into the RastPort being rendered to:

```
#define AREA_BUFSIZE    200 /* space for 40 vectors */

struct AreaInfo areainfo;
struct TmpRas tmpras;
PLANEPTR planeptr = NULL;
WORD areabuffer[AREA_BUFSIZE] = {0};

planeptr = AllocRaster(width, height);
if (planeptr)
{
   InitArea(&areainfo, areabuffer, AREA_BUFSIZE/5);
   rp->AreaInfo = &areainfo;

   InitTmpRas(&tmpras, planeptr, RASSIZE(width, height));
   rp->TmpRas = &tmpras;
}
```

Don't forget to free up the temporary raster on exit with:

```
FreeRaster(planeptr, width, height);
```

## LET'S GET PRIMITIVE

Now that we have some memory to draw into and a RastPort to communicate through, it's time to use some drawing functions. All take pointers to the RastPort into which you want to draw.

First off, you have extensive control over drawing colors and mode; you can even set these individually for multiple RastPorts. There are four standard drawing modes and three drawing pens you have control over—the foreground pen (FGPen or A-Pen), background pen (BGPen or B-Pen), and outline pen (AOlPen or O- Pen). These individual colors are set with the functions:

```
SetAPen(rp, foreground_color)
SetBPen(rp, background_color)
SetOPen(rp, outline_color)
```

You use the outline pen when drawing filled rectangles and areas, as well as when using the Flood() function in mode 0.

The current drawing mode specifies how to draw and with which colors. You set the drawing mode with SetDrMd(). The four possiblities are:

| Mode | Function |
|---|---|
| JAM1 | Draw with the foreground color only |
| JAM2 | Draw with both the foreground and background colors |
| COMPLEMENT | Invert the background |
| INVERSVID | Invert the foreground and background colors |

JAM2 and INVERSVID are most useful when rendering text or patterns. COMPLEMENT is used mostly for "rubber-band" graphics. When you draw the same shape twice in a row, it is erased the second time leaving the background intact. This is demonstrated in the Area program on disk.

You also have control over which planes are rendered to with successive drawing commands via:

```
SetWrMsk(rp, planes)
```

Enabled planes have the appropriate bit set for that plane number (for example, to disable plane two, planes = 0xFB).

You can specify a rendering pattern for both lines and filled areas using SetDrPt(rp, line_pattern) and SetAfPt(rp, area_pattern, power_of_two).

SetDrPt(rp, line_pattern) sets the current line-drawing pattern for the specified RastPort. Line_pattern is a 16-bit word, with a bit set for each pixel to be set in that 16-pixel section of the line. When a RastPort is initialized, this pattern, of course, defaults to 0xFFFF (binary 1111111111111111) for solid lines. To render dotted lines, a value of 0xCCCC (binary 1100110011001100) would do nicely, with two-pixel dots followed by two-pixel blank regions. If you draw multiple lines, the pattern is consistently maintained between segments. To set the RastPort back to drawing solid lines, a value of ~0 (0xFFFF) does the trick.

Note that line patterns can be combined with drawing modes to get two-color lines. If the current drawing mode is JAM2, then the foreground color is rendered where there is a 1-bit set in the pattern, and the background color is used where a bit is set to 0 in the pattern.

An example of using the dotted-line pattern that may not be immediately obvious is the new "lasso-box" under Workbench 2.0. When you hold down the mouse and drag out a region for selecting multiple icons, a dotted-line pattern rotates clockwise as long as you keep the mouse held down. This is accomplished by rendering a box with a dotted-line pattern in COMPLEMENT mode, waiting a few ticks, drawing the box again (which effectively erases it, leaving the background intact), then updating the current line-drawing pattern to one with the dot pattern shifted to the right by one pixel. The process continues, and the pattern appears to move.

You use SetAfPt(rp, area_pattern, power_of_two) to set a pattern that will be used whenever a filled region is rendered, such as a rectangle drawn with RectFill(), a polygon drawn with the area functions, or when using Flood().

Area_pattern is the address of the first word in an array of words that are again 16 bits wide. The array can be any power of two in height (1, 2, 4, 8, 16, 32, and so on). For a 16×16 pattern, power_of_two would be 4 ($2^4$=16). For example, it's easy to set up a very simple crosshatch dither pattern by creating an array of two words, each containing an offset single-pixel dotted pattern:

```
0x5555=0101010101010101 and
0xAAAA=1010101010101010

UWORD crosshatch_pattern[] = {0x5555, 0xAAAA};
SetAfPt(rp, crosshatch_pattern, 1);
```

The one is raised from 2, meaning a height of 2.

Drawing mode also affects how the area pattern is filled. If the drawing mode is JAM1, only pixels with a bit set are drawn in the foreground pen. If the mode is JAM2, then pixels with a bit unset in the pattern are colored with the back-

ground pen as well. JAM2 | INVERSVID would swap the foreground and background pens. Note that the pattern is positioned with respect to the upper-left corner of the RastPort, so as long as your filled regions are adjacent, the pattern will remain continuous.

The area pattern can be set back to a solid fill by specifying NULL as the pattern address, with a power_of_two of 0:

```
SetAfPt(rp, NULL, 0);
```

Areas may also be filled with multicolored patterns. Set up your pattern as an array of area patterns (as above), one pattern per plane in the bitmap, and make power_of_two a negative value. For example, this would set the current area pattern to an eight-color checkerboard pattern for a bitmap three planes deep:

```
UWORD check_pattern[3][8] = {
    /* plane 0 */
  {
    0x0000, 0x0000, 0xFFFF, 0xFFFF,
    0x0000, 0x0000, 0xFFFF, 0xFFFF,
  }
    /* plane 1 */
  {
    0x0000, 0x0000, 0x0000, 0x0000,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
  }
    /* plane 2 */
  {
    0xFF00, 0xFF00, 0xFF00, 0xFF00,
    0xFF00, 0xFF00, 0xFF00, 0xFF00,
  }
};

SetAfPt(rp, check_pattern, 3); /* (2^3=8 pixel high) */
```

It's also a good idea to set the following drawing attributes when using multicolored area patterns:

```
SetAPen(rp, 255);
SetBPen(rp, 0);
SetDrMd(rp, JAM2);
```

To set a specific pixel to the current color, use WritePixel():

```
WritePixel(rp, x, y)
```

To read a pixel value at a certain location:

```
color = ReadPixel(rp, x, y)
```

## TAKE SHAPE

To fill a rectangular region of the BitMap with the current color (or fill pattern, if one is specified):

```
RectFill(rp, xmin, ymin, xmax, ymax)
```

Remember that if the current drawing mode is JAM2, and an area pattern is specified, both the foreground and background colors will be rendered—foreground where there is a bit set in the pattern, background if not. Check out the dropshadow area pattern effect in the Primitives program. The rectangle is always outlined in the current outline pen, unless you use the BNDRYOFF(rp) macro (in graphics/gfxmacros.h) to turn off area outlining first.

*"A drawing mode of JAM2 used with a line-draw pattern set with SetDrPt() will allow you to render two-color patterned lines."*

The following functions draw arbitrary ellipses:

```
DrawCircle(rp, x_center, y_center, radius)
DrawEllipse(rp, x_center, y_center, x_radius, y_radius)
```

To draw simple lines, you can use the following two functions:

```
Move(rp, x, y)
Draw(rp, x, y)
```

If your lines are connected, you do not have to use Move() before each line. Draw() will connect the new pixel with the current pixel, which is stored in the RastPort structure after every graphics rendering. Once again, a drawing mode of JAM2 used with a line-draw pattern set with SetDrPt() will allow you to render two-color patterned lines.

You can also automatically draw a linked set of lines, using PolyDraw():

```
PolyDraw(rp, vertex_count, vertex_array)
```

Create an array with all the vertices of the polygon and pass it to PolyDraw(). This function calls Move() to position the pen at the first vertex and performs successive Draw() operations on the remaining coordinate pairs.

```
WORD triangle[] = {
  10, 10,
  20, 10,
  15, 20,
  10, 10    /* close up the polygon */
};

PolyDraw(rp, 4, &triangle[0]);
```

The following functions allow filled areas to be created quickly. You will need to allocate a temporary raster and initialize an AreaInfo structure to use them, however. They work just like Move() and Draw(), but you use an AreaEnd() to connect the final segment and fill the area. Area pattern and outline pen apply.

```
AreaMove(rp, x, y)
AreaDraw(rp, x, y)
AreaEnd(rp)
```

To draw a triangle, for example:

```
AreaMove(rp, 10, 10);
AreaDraw(rp, 20, 10);
AreaDraw(rp, 15, 20);
AreaEnd(rp);  /* this will close up the area */
```

You can also generate filled circles and ellipses with the routines:

```
AreaCircle(rp, x_center, y_center, radius)
AreaEllipse(rp, x_center, y_center, x_radius, y_radius)
```

To combine some operations, the previous triangle example can be filled with a dither of colors 2 and 4, and outlined in color 5, by prefixing the area operations with the following initializations: ►

```
SetAPen(rp, 2); /* foreground */
SetBPen(rp, 4); /* background */
SetOPen(rp, 5); /* outline */
SetDrMd(rp, JAM2);  /* render with foreground and background */
SetAfPt(rp, crosshatch_pattern, 1); /* dither area-fill */
```

To fill arbitrary outlined regions after they are drawn:

```
Flood(rp, mode, x, y)
```

Mode 0 (outline mode) fills with the foreground color starting at x, y until it encounters pixels drawn in the current outline color. Mode 1 (color mode) samples the pixel at x, y first, and then fills only adjacent pixels of that color. For example, if you have a green polygon and want to color it blue instead, you set the foreground color to blue, and specify any location within the green polygon as x, y (be sure to use 1 for mode).

Simple text rendering (in the current font) is accomplished with the Text() function:

```
Text(rp, text, len)
```

The current Y coordinate specifies where to draw the baseline of the font, so if you want to draw "Hello, World" with the upper-left corner of the H at pixel location 10, 20, you would type:

```
Move(rp, 10, 20 + rp->Font->tf_BaseLine);
Text(rp, "Hello, World", 12);
```

See the Primitives program for examples of how the current colors and drawing modes affect text rendering.

Now add some motion to the screen. To scroll an arbitrary rectangular region of a RastPort, use:

```
ScrollRaster(rp, dx, dy, xmin, ymin, xmax, ymax)
```

The deltas are given in pixels; positive values will scroll the region in the direction of the origin (0,0). The region is scrolled within the bounds provided, so regions outside of the scrolled section are not overwritten. The background color is used to replace areas of the region scrolled. For example, to scroll the region enclosed by 20, 20 and 60, 60 to the left five pixels and down ten pixels:

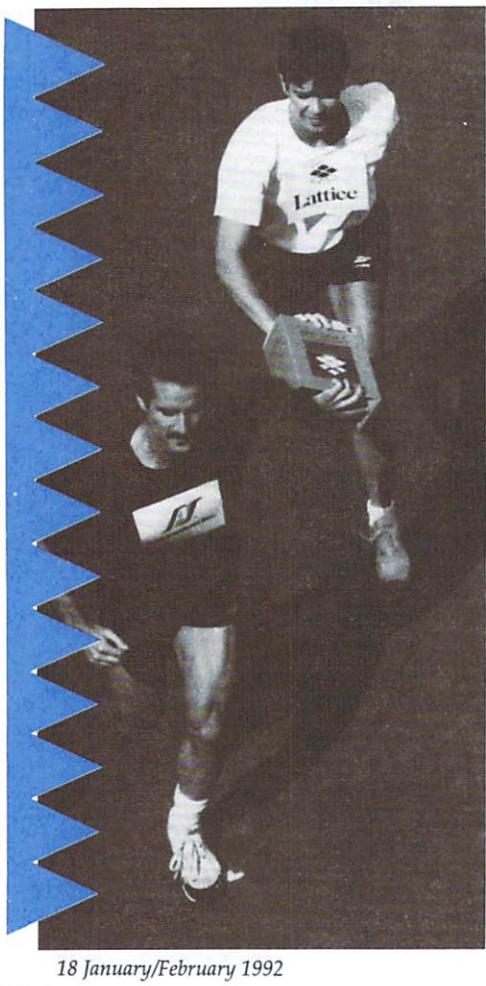
```
ScrollRaster(rp, 5, -10, 20, 20, 60, 60);
```

Remember, the 5 is five pixels towards the origin horizontally, and the –10 is ten pixels away from the origin vertically.

## INFORMATION OVERLOAD

All this flexibility and power is only the tip of the graphics.library iceberg—there's still bitmaps, text, sprites, BOBs, and animation, as well as Copper-list tricks and special display modes. We'll delve into these in future articles. For now, play around with the two sample programs, especially Area, which also has a good deal of Intuition and interactive polygon algorithms. ■

*Paul Miller has been a developer since 1985 and is currently majoring in Computer Science at Virginia Tech. Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or via Internet (pmiller@vttcf.cc.vt.edu).*



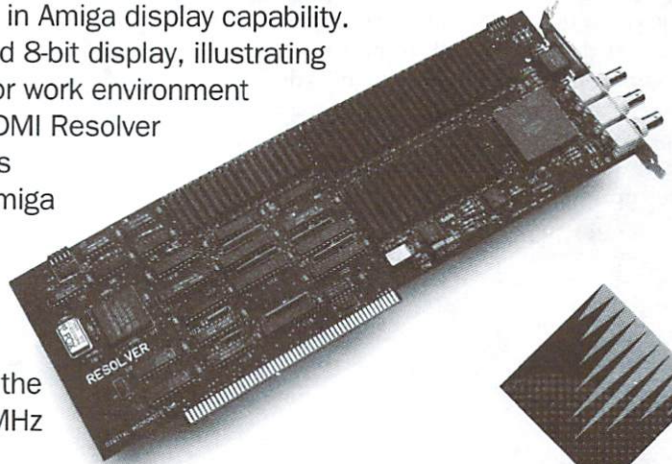
# Don't be Fooled by any other Solution. 1280x1024 Resolution.

## DMI Resolver™

- 1280x1024 Resolution
- 8-bit Color Graphics
- 16-million Color Palette
- 60MHz Processor
- Programmable Resolution

The DMI Resolver™ graphics co-processor board offers a new dimension in Amiga display capability. Shown above is an unretouched 8-bit display, illustrating the 1280x1024 resolution color work environment provided by the Resolver. The DMI Resolver boosts the display and graphics processing capabilities of all Amiga A2000 and A3000 series computers, under both AmigaDOS and UNIX operating systems. Not to be confused with a frame buffer or grabber, the Resolver is a lightning fast 60MHz graphics co-processor.

Whatever your application – desktop publishing, presentation graphics, animation, 3D modeling, ray tracing, rendering, CAD – let the Resolver move you into a new realm of resolution and workstation quality display.




**Digital Micronics, Inc.**
5674 El Camino Real, Suite P
Carlsbad, CA 92008
Tel: (619) 431-8301 • FAX: (619) 931-8516
Call for more information and the dealer nearest you.



Circle 6 on Reader Service card.

# Faster 3-D Drawing

ON DISK

By Brian Carmichael

THE MATHEMATICAL DERIVATION required to display 3-D objects on an Amiga is reasonably simple. The "text book" implementation, however, normally uses floating-point math. Fine, if you have floating-point coprocessors, but if you must rely on software emulation this approach suffers from slow execution speed. After we examine how to derive a general solution for a 3-D display routine, I will demonstrate how implementing the same routine using integer arithmetic gives you an up to 60% increase in performance.

## DERIVATION OF THE 3-D TRANSFORMATION

The goal of a 3-D transformation is to convert objects described in three-dimensional coordinates to a form suitable for display on a two-dimensional computer screen. You can perform such a transformation in two steps. First, orient the 3-D object coordinates with respect to the display screen. Next, convert the screen-oriented 3-D coordinates to 2-D screen coordinates using a perspective transformation.

3-D objects are traditionally represented on the display screen with the Y axis running up and down the screen, the X axis running from left to right, and the Z axis running "into" and "out of" the screen. The transformation from a 3-D object to this screen representation is shown in Figure 1.

For a proper display, you must also be able to specify the position and orientation of the viewer with respect to the 3-D object space. In Figure 2, the viewer's focus point is defined as point F. The vector from point C, the camera point, to the focus point, F, specifies the viewing direction. The vector from point F to point U defines the direction that the viewer considers to be "up." The three points, C, F, and U, constitute the "view."

The first step in constructing the 3-D transformation is to move, or translate, the view's focus point to the origin. See Figure 3. Each point in the view is shifted towards the origin using the operation:

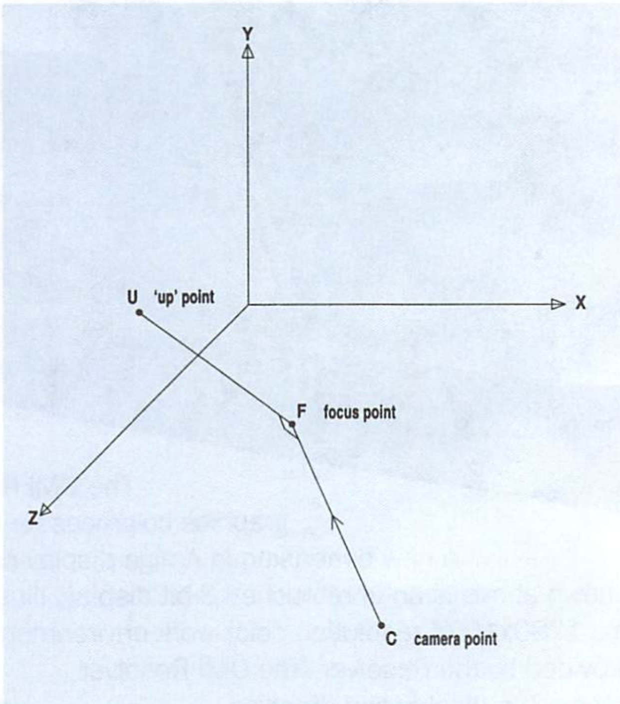



Figure 2. The definition of a view.

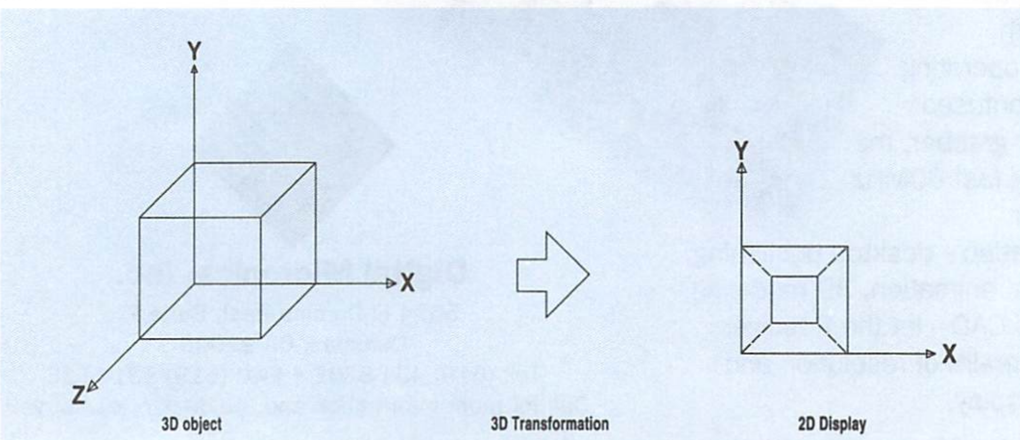



Figure 1. A 3-D to 2-D transformation.

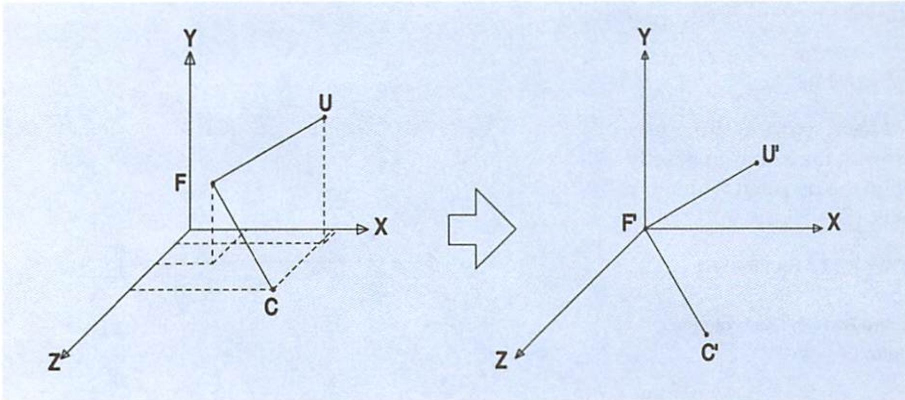



Figure 3. Translate the view's focus point to the origin.

$P' = P * T$

**where: P is the origin position of the point**
**P' is the translated position**
**T is the 4x4 translation matrix**

$$T = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Fx & -Fy & -Fz & 1 \end{vmatrix}$$

**and [Fx Fy Fz 1] are the 3-D coordinates of the focus point F.**

After this operation, the translated view points have the values:

$F' = [\,0\;0\;0\;1\,]$
$C' = [\,Cx - Fx, Cy - Fy, Cz - Fz, 1\,]$
$U' = [\,Ux - Fx, Uy - Fy, Uz - Fz, 1\,]$

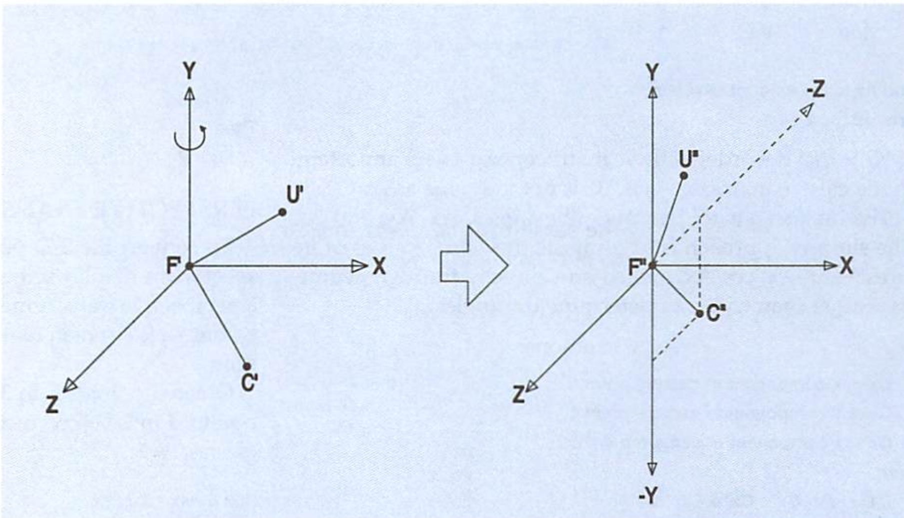



Figure 4. Rotate the view about the Y axis to align the camera point with the minus-Z-X plane.

For the second step, you rotate the view about the Y axis to align the camera point with the minus-Z-X plane. Shown in Figure 4, the transformation required is:

$P'' = P * T * Ry$

**where Ry is the 4x4 rotation matrix**

$$Ry = \begin{vmatrix} \cos Ay & 0 & -\sin Ay & 0 \\ 0 & 1 & 0 & 0 \\ \sin & Ay\ 0 & \cos Ay & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

**and Ay is the amount to rotate about the Y axis**

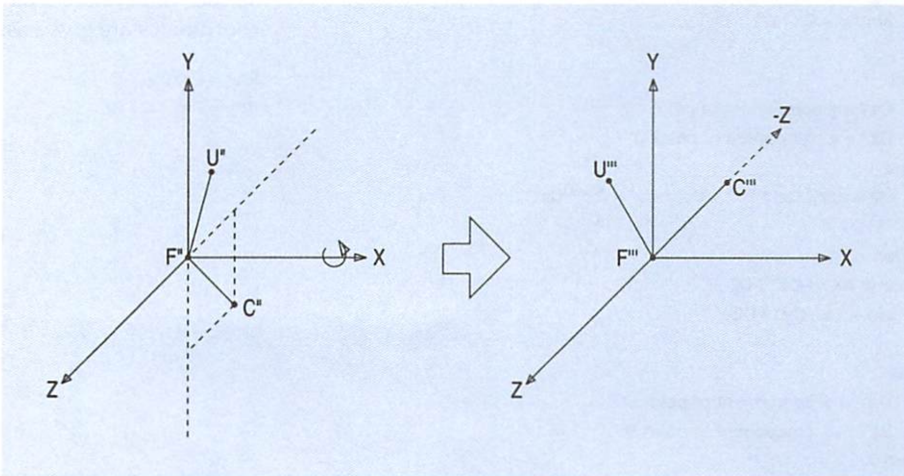



Figure 5. Rotate the view about X axis to align the camera point with the minus-Z axis.

After the second step, the points have the values:

$F'' = F'$
$C'' = C' * Ry$
$U'' = U' * Ry$

In the third step, you rotate the view about the X axis to align the camera point exactly with the minus-Z axis (Figure 5).

$P''' = P * T * Ry * Rx$

**where Rx is the 4x4 rotation matrix**

$$Rx = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos Ax & \sin Ax & 0 \\ 0 & -\sin Ax & \cos Ax & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

**and Ax is the amount to rotate about the X axis**

The view is now given by:

F''' = F'
C' '' = C'' * Rx
U''' = U'' * Rx

Next, rotate the view around the Z axis to exactly align the up point with the Y axis. (See Figure 6.)

P'''' = P * T * Rx * Ry * Rz

where Rz is the 4x4 rotation matrix

$$Rz = \begin{vmatrix} \cos Az & \sin Az & 0 & 0 \\ -\sin Az & \cos Az & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

and Az is the amount to rotate around the Z axis

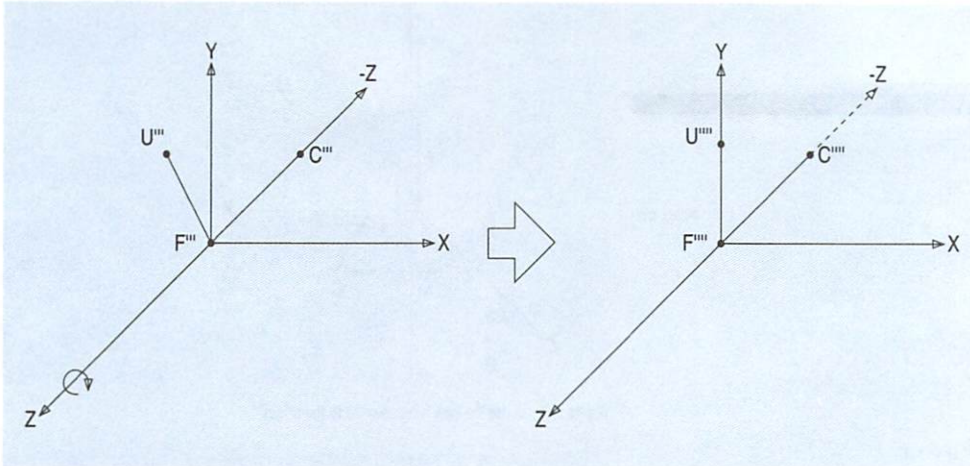



Figure 6. Rotate the view around the Z axis to align the up point with the Y axis.

Note that the order of these matrix operations is important! In the case of matrices, A=B * C is not the same as A=C * B.

The final step is to determine the angles, Ax, Ay, and Az. The simpler approach is to compute the actual values of interest—sin Ax, cos Ax, and so on—directly through geometry, rather than trying to determine the angles.

Let
Cx' = x component of camera point C'
Cy' = y component of camera point C'
Cz' = z component of camera point C'
and
D1 = sqrt( Cx' * Cx' + Cz' * Cz' )

Then
cos Ay = - Cz' / D1
sin Ay = Cx / D1

Let
Cy'' = y component of point C''
Cz'' = z component of point C''
and
D2 = sqrt( Cx' * Cx' + Cy' * Cy' + Cz' * Cz' )

Then
cos Ax = –Cz'' / D2
sin Ax = –Cy'' / D2

Let
Ux''' = x component of point U'''
Uy''' = y component of point U'''
and
D3 = sqrt( Ux''' * Ux''' + Uy''' * Uy''' )

Then
cos Az = Uy''' / D3
sin Az = Ux''' / D3

A point, P, can be converted from 3-D object space to the 3-D screen orientation, Ps, using a combined transformation matrix, M, as follows:

Let
M = T * Ry * Rx * Rz

Then
Ps = P * M

## PERSPECTIVE TRANSFORMATION

To convert the 3-D points that are now oriented with respect to the display screen to 2-D screen coordinates, you use a perspective transformation. In Figure 7, objects are projected onto a flat screen that is located distance D from the focal point.

Given a point, Ps, in 3-D screen-oriented coordinates, the point Pd in 2-D screen coordinates is given by similar triangles as:

Pdx = Psx * D / Psz
Pdy = Psy * D / Psz

Alternatively, you can project the points onto a sphere's surface rather than a flat plane. For this case, the 2-D screen coordinates are given as:

Pdx = Psx * D / R
Pdy = Psy * D / R

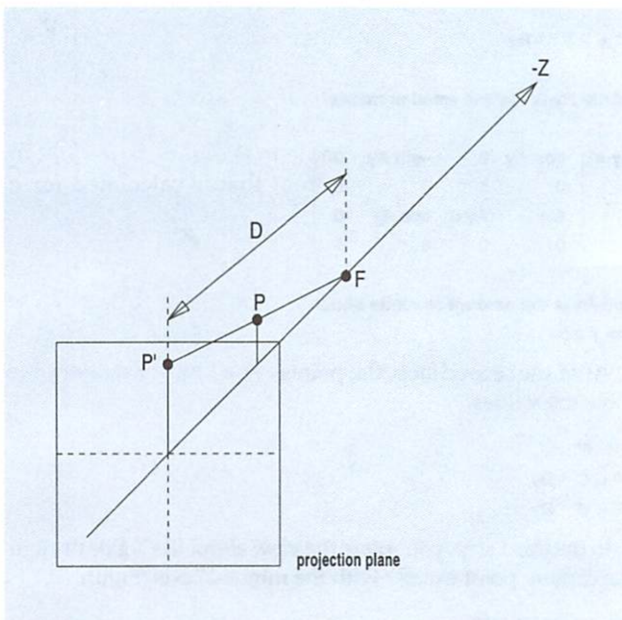



Figure 7. A perspective projection.

where R = sqrt( Psx * Psx + Psy * Psy + Psz * Psz )

Therefore, you convert a point, P, in 3-D object coordinates to 2-D coordinates as follows:

```
Let
   P  = point in 3-D object space
   Ps = 3-D point in screen-oriented coordinates
   x, y = 2-D screen coordinates

Then
   Ps = P * T * Ry * Rx * Rz
      = P * M
   x = Psx * D / Psz
   y = Psy * D / Psz
```

## PRACTICAL CONSIDERATIONS

The above theory sounds clean and simple, but practical implementation requires you to keep in mind a few cautions. First, the viewer cannot look straight up or down, because the first rotation (the one about the Y axis) is then indeterminant. Second, you cannot perform a perspective transformation when z equals 0, as this is also indeterminant. Finally, the mathematics of the transformation does not preclude the calculation of points that would be considered not displayable normally, such as points situated behind the viewer. This means you need a method of clipping away 3-D points from the object, unless you are guaranteed that all its points will lie in front of the viewer. Even in this unlikely case, points situated close to the focal point may be magnified to such an extent that they lie well outside the screen bounds.

*"Circle clipping is a good choice because it is simple to program."*

You have your choice of clipping methods. One is to reject any 3-D point that lies outside a 3-D "viewing-volume." In an alternate approach, known as circle clipping, you use the distance to the origin from the 3-D point to be clipped as the radius of a circle. You then project this circle onto the viewing surface. If the circle lies outside the screen boundaries, you reject the point. Circle clipping is a good choice because it is simple to program, and you can use the intermediate square-root that is calculated for a spherical projection.

To understand the basics of circle clipping, study the Slow program in the accompanying disk's Carmichael drawer. Implementing the above algorithm in floating point, the program takes the following arguments:

```
slow -o object-file -c camera-file -n
```

The –o specifies an ASCII file containing 3-D coordinates of an object in the form

```
m x y z
d x y z
```

where m specifies a 3-D move operation and d specifies a 3-D draw operation.

The –c specifies an ASCII file containing 3-D coordinates of a camera or viewing file in the form

```
cx cy cz fx fy fz
```

where (cx cy cz) are the 3-D coordinates of the camera point and (fx fy fz) are the 3-D coordinates of the focal point.

The –n specifies that no drawing is to be performed, just the 3-D to 2-D calculations.

If an argument or arguments are not given, camera-file defaults to camera.def and object-file to object.def.

Slow reads in the camera and object data and stores them internally. It then creates a 600X200 window to display the results. For each view in camera-file, Slow displays the object defined by the points in object-file. The program terminates after it cycles through all the views and the user selects the window's close gadget. The total time in seconds to compute and display the 3-D object is then shown.

Slow consists of the following source files:

| | |
|---|---|
| main.c | reads in data, performs controlling actions |
| persp.h | contains definitions, function prototypes, data structures |
| graphic.c | handles all machine-dependent graphics routines |
| slow.c | performs 3-D transformation routines |

Remember two things as you study the source. While Slow uses a spherical projection surface by default, you can change the definitions in persp.h and recompile slow.c to project onto a flat plane. Only two points are used to specify the view. In this program, up is always the positive Y axis. The setup_matrix routine in slow.c can handle three-point views, but here I hard-coded up to be the positive Y axis.

## SPEED ADJUSTMENTS

While functional, Slow can be sped up in many ways. First, note what happens when up is specified as always the positive Y axis. The angle of rotation, Az, is then always 0. The rotation matrix, Rz, is then the identity matrix, I, (with all 1s in the diagonal). This means that:

```
M = T * Ry * Rx * Rz
  = T * Ry * Rx * I
  = T * Ry * Rx
```

Therefore, the Z-axis rotation matrix, Rz, can be eliminated completely. As in a flight simulator program, Rz controls the amount of bank or tilt of the 3-D objects viewed on the screen. If you wish, however, you can still accomplish this effect by performing a 2-D rotation of the screen coordinates afterward.

Matrix T has the effect of a simple subtraction of the focus point from each 3-D point. It is simpler to break the transformation into two steps:

```
1) P' = P * T
      = P - F

2) Ps = P' * Ry * Rx
      = P' * R
```

The combined rotation matrix, R=Ry * Rz, can be determined by substituting the values for sin Ay, cos Ay, sin Ax, and cos Ax (which were calculated earlier) into matrices Ry and Rx. ►

$$Rx = \begin{vmatrix} -Z/D1 & 0 & -X/D1 & 0 \\ 0 & 1 & 0 & 0 \\ X/D1 & 0 & -Z/D1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

$$Ry = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & -B/D2 & -Y/D2 & 0 \\ 0 & Y/D2 & -B/D2 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

```
where
   X = Cx – Fx
   Y = Cy – Fy
   Z = Cz – Fz

   D1 = sqrt( X * X + Z * Z )
   D2 = sqrt( X * X + Y * Y + Z * Z )

   B = –( X * X + Z * Z ) / D1
```

Multiplying together and simplifying results in:

R = Ry * Rx

$$= \begin{vmatrix} xx & yx & zx & 0 \\ 0 & yy & zy & 0 \\ xz & yz & zz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

```
where
   xx = –Z/D1
   xz = X/D1
   yx = (–X * Y) / (D1 * D2)
   yy = –B/D2
   yz = (–Z * Y) / (D1 * D2)
zx = (X * B) / (D1 * D2)
zy = –Y / D2
zz = (Z * B) / (D1 * D2)
```

You can then perform the 3-D transformation as:

```
Let
   P = 3-D point

   X = Px - Fx   ; result of matrix T
   Y = Py - Fy
   Z = Pz - Fz

   D = projection distance
   Z2 = depth of point

   (Sx, Sy) = resultant screen coordinate point
```

Then for projection onto flat surface:

```
Z2 = X * zx + Y * zy + Z * zz

Sx = ( X * xx + Z * xz ) * D / Z2
Sy = ( X * yx + Y * yy + Y * yz ) * D / Z2
```

Examination of matrix R = Ry * Rz reveals that the range of possible values in each entry is between –1 and 1. This is an ideal candidate for implementation using fixed-point numbers. You can use a 16-bit number of the form:

```
S.BBB BBBB BBBB BBBB

where
   S is a sign bit
   B is a mantissa bit
```

The 16-bit result of a multiplication of a 16-bit coordinate point, P, and a 16-bit fixed point, F, in the above format is then:

```
R = ( P * F ) ASR 15

where
   * is a normal signed integer multiply
   ASR is the arithmetic-shift-right (sign-0preserving) operator
```

Note that 16-bit values are used for P and F because the intermediate value (P * F) will then fit entirely within a 32-bit LONG integer.

For more speed, you can use a power of 2, such as 512, for the projection distance, D. This replaces a slow multiplication operation with a faster shift operation.

Clipping is performed as soon as Z2 is calculated. You can save time by computing Sx and Sy only if the 3-D point is not clipped.

*"For more speed, you can use a power of 2, such as 512, for the projection distance, D."*

### FASTER STILL

The 3-D algorithm can be implemented using entirely integer arithmetic via this fixed-point notation, as the Fast program (in the Carmichael drawer) demonstrates. Aside from the simplifications mentioned above and execution speed, Fast is similar to Slow, with the same arguments and behavior. Fast performs circle clipping as Slow does. The operation involves the calculation of a square root once per 3-D pixel. Fast uses Komusin's integer square-root routine (see the references list) instead of performing a floating-point calculation. All of the code is implemented in C except for the square-root routine. Assembly language would have been faster, but I wanted to make this program as readable as possible.

Fast consists of the source files:

| | |
|---|---|
| main.c | same as for Slow |
| graphic.c | same as for Slow |
| persp.h | same as for Slow |
| isqrt.asm | Komusin's integer square root routine |
| fast.c | fixed-point version of Slow |

Two routines in fast.c are all that is needed to handle the fixed-point integer manipulations. FracToLong converts a given integer numerator and denominator into a 16-bit fixed-point number, while fixedMult multiplies an integer by a fixed-point number and returns an integer. In assembly, this could be coded simply as:

```
; d0 = 16 bit integer
; d1 = 16 bit fixed-point number

muls  d1,d0   ; d0 = 32-bit result of d0 * d1
asr.l  #8,d0
asr.l  #7,d0   ; d0 = 16-bit normalized result
```

The code to perform the same action in C is more involved, because the C shift-right operator, >>, destroys the sign. The code in fixedMult takes pains to preserve the sign of the result.

Fast.c uses long values even for the 16-bit fixed-point numbers, so that the compiler will not generate time-wasting code to convert SHORTs to LONGs and vice-versa.

As proof of the improvement's success, study the comparative-timings chart below. I timed Slow and Fast on my veteran Amiga 1000 with a 16-MHz 68020 and four megabytes of 32-bit RAM. I used the Fast Floating Point (FFP) library for all floating-point operations. Timings are given as mm/nn seconds, where mm is the time to calculate and display results exclusive of all set-up time, and nn is the calculation time alone (–n option). In all cases I used the camera file camera.def.

**TABLE 1**

| object file | Slow time | Fast time | Performance Improvement |
|---|---|---|---|
| xyz.def | 8/5 | 5/1 | 38 % |
| object.def | 47/33 | 25/10 | 47 % |
| spiral.def | 70/47 | 32/10 | 54 % |
| cone.def | 57/48 | 23/11 | 60 % |

Significant time savings, 50% on average, were realized by the integer-based version over the FFP-based version. Note that a significant amount of time, 47% on average, is spent on drawing and other overheads exclusive of the 3-D calculation time.

### TIME OUT FOR FUN

The program Fly (in Carmichael) is a primitive "flight simulator" that lets the user move around a 3-D world using the joystick. To invoke Fly, type:

```
Fly –o object-file
```

In Fly, object-file has the same format as in Slow and Fast. Pushing the joystick forward or back moves you forward or back with respect to the 3-D object. Pushing left or right causes left and right turns. You terminate the program with the joystick button. The square "compass" in the middle of the screen will help you orient yourself in the 3-D world.

The source files that comprise Fly are:

| | |
|---|---|
| fly.c | main program |
| graphic.c | as per Slow and Fast |
| isqrt.asm | as per Fast |
| fast.c | as per Fast |
| joy.c | joystick routines |

*"The program Fly is a primitive 'flight simulator' that lets the user move in a 3-D world using the joystick."*

One caution: Joy.c violates proper Amiga programming practices by examining the joystick port hardware directly. The lapse is best explained (but not excused) by too many late-night pizzas. Do not copy this technique.

### TWEAK IT YOURSELF

Undoubtedly, Fast would be faster written in assembly language. The fixedMult routine, for one, would be much speedier. If you do not want to translate it to another language, however, you can implement the suggestions below to improve Fast.

• The Z rotation matrix was eliminated by making the up direction always the Y axis. You could extend this idea by not performing the X rotation or Y rotation. Eliminating both leaves a simple perspective transformation that might be perfectly suitable for, say, a road-race game.

• You could eliminate the flicker in the display window with double buffering. In the interest of simplicity, the example programs merely erase the background before drawing the next view, resulting in noticable flicker.

• Transform only one point per 3-D object. The object is rendered on the screen using the transformed key point to provide location and scaling information.

• The example programs perform clipping by removing the entire offending line segment. This can result in the mysterious disappearance of long ling segments as you move around in the 3-D world. You can achieve a slower, but more pleasing, effect by recursively subdividing line segments that cross the clipping boundary until you find a subsection that can be displayed.

• Add a solid-object display option. Brian Wagner's article (see the reference list) provides good instructions.

Once you understand the basic algorithm, alterations should be a cinch. ■

*Brian Carmichael is a software engineer for Imapro Corp., makers of electronic prepress equipment, and has been programming the Amiga for four years. Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

## References

Foley, J. D. and Van Dam, *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, 1984.

Watt, A., *Fundamentals of Three-Dimensional Computer Graphics*. Addison-Wesley, 1989.

Komusin, B., "Fast Algorithm Computes Square Root," *EDN*, Nov. 1987.

Newton, M., "Real-Time 3-D Graphics for Microcomputers," *BYTE*, Sep. 1984.

Wagner, B., "Building a 3-D Object Viewer," *The AmigaWorld Tech Journal*, Jun./Jul. 1991.

ON DISK

# ARexx is for Writing Applications, Too

By Marvin Weinstein

ALL OF US HAVE HEARD the parable of the three blind men who, confronted for the first time by an elephant, attempt to describe the fabulous beast. The first blind man, taking hold of the elephant's trunk, declares "An elephant is like a snake." The second blind man, running his hands over the elephant's side, avows "An elephant is like a wall." Finally, the third blind man, grabbing hold of the elephant's massive leg, asserts "An elephant is like a tree."

Recently, when I overheard someone answer the question "What is ARexx?" by saying it is Commodore's interprocess communication language, I realized how much this parable applies to the way people currently view REXX on the Amiga. While this, and the other popular response, "ARexx is Commodore's universal scripting language," are true, they no more describe ARexx than the blind men described the elephant.

ARexx is a realization of the REXX programming language as defined in M.F. Cowlishaw's book *The REXX Language, A Practical Approach To Programming* (Prentice-Hall, 1985). Although it eventually replaced IBM's EXEC2 scripting language, REXX was not created for that purpose alone. From the outset REXX was meant to be an eminently usable high-level programming language. Most of its unique features—the lack of typed variables, the extensive library of functions to manipulate strings, the excellent built-in interactive debugging tools—follow directly from the usability requirement. The extent to which REXX has become an indispensable part of the VM/CMS working environment shows that this goal was achieved.

> *"When Bill Hawes created a realization of ARexx for the Amiga, he gave it powers far beyond those of mainframe REXX."*

If REXX is not just another scripting language, then ARexx is not just a port of REXX to the Amiga. When Bill Hawes created a realization of REXX for the Amiga, he gave it powers far beyond those of mainframe REXX. AmigaDOS is a true multitasking operating system, and Hawes made certain that ARexx could take full advantage of that fact. He also made it easy to access the Amiga's mechanisms for passing messages between applications. These capabilities positioned ARexx to be the universal Amiga scripting language and explain why it has become fashionable to refer to ARexx as the Amiga's interprocess communication language. That Hawes accomplished all this without compromising REXX's usability or extensability is a tribute to his skill as a software designer.

Like the elephant, ARexx cannot be described by focusing on just one of its attributes. ARexx the universal scripting language, ARexx the interprocess communication language, ARexx the prototyping language, and ARexx the high-level programming language are all just facets of this marvelous beast. Unfortunately, the high-level programming aspect is too often overlooked, or even denied, by people used to programming in BASIC or C. Misled by ARexx's typelessness, they consider it unsuitable for writing complicated, stand-alone applications. In this article I'll show, by example, that ARexx is well suited to writing complicated applications, provided they are not too computationally intensive. This limitation exists because ARexx is not a compiled language.

### THE CHALLENGE

As proof that you can write a useful, stand-alone application in ARexx, I offer a reminder and calendar utility. This type of utility, commonly sold as a commercial product, needs certain features. First, it must be Intuitionized—setting a reminder must be essentially a point-and-shoot procedure. Second, the text of a reminder must appear, at the designated time, in a window that stays open until the user closes it. Third, there must be a nag feature, comparable to the snooze button on an alarm clock. Finally, it must be possible to schedule a command for execution at the time the reminder is displayed.

### BASIC DESIGN DECISIONS

The reminder utility consists of several programs. Of these, the biggest one creates and manages the custom requester used for setting reminders. All the other programs take care of housekeeping details. The custom requester has button gadgets labeled Date:, Set Hours, Set Minutes, AM (or PM), Nag Me, Say It, and Add Reminder; it also has several string gadgets. Clicking on the Date: gadget brings up a calendar for setting the date the reminder will appear.

When the calendar closes, this date automatically enters the string gadget located to the right of the Date: gadget. It is possible to avoid using the calendar by typing a date directly into this string gadget. The next three gadgets, labeled Set Hours,

*There's more to ARexx than interprocess communication, as this calendar/reminder program proves.*

Set Minutes and AM (or PM), are for setting the reminder's time of day. Clicking on the Nag Me gadget sets the reminder as a nag-type, clicking on the Say It gadget specifies that the reminder is to be both spoken and displayed, while clicking on the Add Reminder gadget schedules the reminder and saves it to disk.

A reminder's window can be closed in two ways. The first is to click on the close gadget in the upper-left corner of the window; the second is to click on the gadget labeled Okay! Okay!. If the reminder is not a nag-type, either operation will make the window disappear permanently. If the reminder is a nag-type, clicking on the close gadget will make it disappear for a specified interval to give the user time to finish what he is doing, then the message will come up again. Nag-type reminders keep reappearing until the user clicks on the Okay! Okay! gadget, at which point they vanish permanently.

The calendar gadget, for setting the date of a reminder, is a separate program, to make the reminder program less complicated. Of course, an independent calendar utility is useful to have. When the calendar program is launched, it displays the current month and highlights the current day. There are button gadgets for resetting the month and year, and a menu item for jumping to a distant date without "mousing around."

Once a reminder has been created it must be stored on disk because it has to survive the computer being turned off. A command can be added to the start-up sequence for rescheduling current reminders and erasing expired ones whenever the Amiga is rebooted. For the user who never turns off his Amiga, I've included a utility that automatically schedules the next day's reminders each time the system clock strikes midnight.

## INSTALLING THE CALENDAR UTILITY

Given the size of the calendar and reminder programs, it is impossible to treat both in a single article. The remainder of this article deals solely with the calendar utility; the reminder utility will appear in the next issue.

The calendar program is located in the calendar.rexx file found in the Weinstein drawer of the companion disk. A second program, testcalendar.rexx, shows how to use calendar as an input device. Copy both of these files to your rexx: directory. Also, because the calendar program uses functions found in rexxarplib.library, you must copy this file and arp.library into your libs: directory. Run the calendar program by typing:

```
rx calendar
```

A window should open in the upper-left corner of your screen. This window contains close and front/back gadgets in its top border and five button gadgets at the bottom. Four of these gadgets, arranged in a square, are for changing the month and year. Clicking on a gadget marked with a plus or minus sign changes the string located to its right. Once these gadgets have been used to select the month and year, click on the Redo gadget to update the display. The calendar window has an menu attached that contains a single item labeled Jump To. Selecting this item brings up a string requester for the month (type in a number between 1 and 12) and the year. Hitting return updates the display automatically. This capability is useful for going directly to a distant date.

To see the use of the calendar window as an input device, type:

```
rx testcalendar
```

This program launches the calendar utility and waits for it to report the day selected by the user. To select a date, position the mouse pointer over the desired day and click on it. If necessary, use the button gadgets to change the month and year. Be sure to update the display before selecting a date. Once a date is chosen, or the close gadget is clicked on, the calendar window will disappear and testcalendar will post the message it receives. This program demonstrates communication between cooperating ARexx programs.

## VERSIONS OF AREXX

The version of ARexx distributed with AmigaDOS 2.0, or as a stand-alone product by Bill Hawes, has certain enhancements over previous versions. When writing this calendar utility, I had to decide whether to use these new features or stay compatible with older versions of ARexx. I opted to use these features, because they are in the Commodore-sanctioned version of ARexx and the cost of an update is so reasonable. I've provided a second version of the calendar program, altcalendar.rexx, for users who don't have a version of ARexx that supports the new features of the PRAGMA() and DATE() functions. The work-arounds used in the alternate version are unnecessarily complicated, and I urge you to update as soon as possible. Current version numbers for the ARexx libraries are rexxarplib.library 2.96 and rexxsupport.library 34.9.

## OVERVIEW OF THE PROGRAM

As you read the following description, refer to the listings calendar.rexx and testcalendar.rexx. Material omitted from ►

the text of the article because of space limitations appears in the listing in the form of comments. Because my previous article, "Custom Interfaces With ARexx" (p. 28, December '91), deals extensively with using rexxarplib to create a rexxarplib host and custom requester, I only touch upon these points here. A full description of rexxarplib commands is in the rexxarplib.doc file on the companion disk.

Several points related to stem variables and features of ARexx functions cannot easily be relegated to comments in the program listings. I conclude this article with a discussion of these matters.

The main body of the calendar program looks like:

```
/** rexx:calendar.rexx
*
**/
call addlib('rexxarplib.library',0,-30,0)
parse arg replyport
hilite = 0
dhnum = PRAGMA(Id)
hostname = DATEHOST||dhnum
portname = DATEPORT||dhnum
call SetDefs()
call GetCurrentDate()
wintext = CalendarText()
call OpenCalendarWindow()
call HiLiteDay(DOIT)
do forever
   if quitflag = 1 then leave
   t = waitpkt(portname)
   do ff = 1
      p = getpkt(portname)
      if c2d(p) = 0 then leave ff
      command = getarg(p)
      parse var command command x y .
      t = reply(p, 0)
      select
         when command = CLOSEWINDOW then
         do
            call CloseWindow(hostname)
            quitflag = 1
            if replyport ~= "" then ,
               interpret address replyport CONTINUE
         end
            .
            .
            .
         otherwise nop
      end
   end
end
exit
```

The initial call to ARexx's addlib() function is included as a precaution, in case the user doesn't do this in his startup-sequence. It is safe to make this call, because addlib() adds a name to ARexx's library list only once. The next line uses ARexx's PARSE instruction, followed by the keyword ARG, to set the variable replyport equal to the command-line argument used when the program was invoked. The replyport argument is optional. If supplied, it indicates that the user is calling the calendar as an input device and defines the port to which a message reporting the result of the user's actions should be sent. ARexx's PARSE instruction is a flexible and powerful tool, and is used several more times, for example:

```
parse var command command x y .
parse upper var x mymonth myyear .
parse upper var xx month"/"day"/"year
```

As the examples indicate, the PARSE instruction extracts one or more substrings from an input string and assigns them to ARexx variables. This instruction, such an important tool in writing ARexx programs, needs further discussion. The generic form of the PARSE instruction is:

```
PARSE [ UPPER ] inputsource [ template ] [ , template ...]
```

*"The PARSE instruction extracts one or more substrings from an input string and assigns them to ARexx variables."*

The optional keyword, UPPER, instructs ARexx to convert all of the extracted strings to uppercase automatically. There are many sources for the input string. It can be a command-line argument or the contents of an ARexx variable. It can be something the user types at the console, or it can be information provided by an ARexx process. Placing the keyword VAR before the names command, x and xx, indicates that the source is an ARexx variable.

The template that follows specifies how to break up this string and assign it to variables. The first example divides the string command into four substrings, using spaces to delimit words. It then assigns the first word to the variable command, the second word to the variable x, the third word to y, and the rest to the period (.) placeholder symbol. Using the . in a parse template does not make an actual assignment; it discards a substring that would have gone there.

A placeholder symbol at the end of a parse instruction is useful, even if you know the number of words in the variable being parsed, because REXX includes the space before the last word when assigning it to the corresponding argument in a parsing template. By adding the . at the end of the template in the first example, we guarantee that there is no leading space in the string y.

The final example uses a more complicated parsing template that separates the variable xx, which contains a date in the form 08/22/91, into three variables, using the / as a separator. The quotes indicate the character, or characters, to be used as separators. This example results in the value 08 being assigned to the variable thismonth and the value 22 to the variable thisday, and so forth. Input sources other than ARG and VAR may be specified with the keywords PULL, NUMERIC, SOURCE, VALUE ... WITH, and VERSION. See the ARexx manual for explanations of these terms.

Warning: A common problem arises when using parsing templates in that REXX attempts to treat expressions enclosed in quotes and followed by an isolated x (or b) as hex (or binary). Thus, a parsing template of the form

```
parse var xx a"/"x"/"c
```

will produce the error message

```
++ Error 8 in line 1: Unrecognized token
```

This will also happen if you replace the x with b.

Back to the calendar listing, the call to ARexx's PRAGMA() function generates a unique name for the calendar's messageport and rexxarplib host. This lets the same program be run several times in order to display multiple calendars. Although the PRAGMA() function has been around for some time, early versions of it recognized only two keywords, Directory and Priority. The most recent version of ARexx adds the option Id. The call PRAGMA(Id) returns the task ID associated with the current invocation of the program, which can be used to uniquely identify the program.

The next four lines of code are calls to subroutines that appear after the main body of the program. SetDefs() defines two ARexx stem variables, months. and daynames., that are used in do loops to construct text strings. Stem variables are REXX's form of arrays and allow for powerful, flexible indexing conventions. In this program I make little use of this flexibility, because I only define arrays of the form:

```
months.1 = 'January'
months.1.days = 31
.
.
.
months.3 = 'March'
months.3.days = 31
```

This format is self-explanatory. An array consists of a stem name, followed by a dot, followed by an index, which can be a number or a string. Additional index variables can be added by preceding the new number or string with another dot. Once an array is defined,

```
j = 3
say months.j
say months.j.days
```

will produce

```
March
31
```

As with all REXX variables, unless the programmer indicates otherwise, stem variables are uppercased automatically when they are defined. Thus, months.1.days is in reality MONTHS.1.DAYS. Failure to remember this can result in confusion, because it is easy to err when using a REXX variable to access the contents of a stem variable. For example,

```
foo = "Days"
goo = "DAYS"
say months.1.foo
say months.1.goo
interpret say months.1.upper(foo)
```

produces

```
MONTHS.1.Days
31
31
```

Observe that I have used REXX's INTERPRET instruction to force the evaluation of upper(foo).

Note that, as with other ARexx variables, unassigned stem variables automatically get a value of their uppercased name. Assuming that no assignment has been made to the variable foo.i.stemvariable, typing

```
say foo.i.stemvariable
```

will produce

```
FOO.I.STEMVARIABLE
```

Because stem variables are always defined, there is no built-in way to know how many values an array contains. It is up to you to provide this information. Also, because index variables can be arbitrary strings, it is up to you to adopt methods for looping over items in a general stem array. Although there are subtleties associated with using stem variables, my use of the stem-variable mechanism here is entirely straightforward.

*"As with all REXX variables, unless the programmer indicates otherwise, stem variables are uppercased automatically when they are defined."*

All manipulations required to get the current day and date are isolated in the GetCurrentDate() subroutine. Because this routine uses ARexx's DATE() function, which is not completely documented in the ARexx manual, the full syntax of the DATE() function appears in the listing.

CalendarText() returns a formatted string that passes to rexxarplib's WindowText() function. This subroutine is adapted from a CLI-based calendar program written by Mike Meyer and distributed on an early ARexx disk.

The remaining subroutines take care of opening the calendar window and highlighting the current day. OpenWindow() to the program discussed in "Custom Interfaces With ARexx." I refer you to it should you need additional information. The main loop, which handles the incoming messages, is unremarkable; however, note the line

```
if replyport = " " then interpret address replyport CONTINUE
```

which sends a message to the ARexx program that called the calendar as an input device. The INTERPRET command in front of ADDRESS replyport CONTINUE forces evaluation of the variable replyport before the message is sent. The same thing could have been achieved without the INTERPRET instruction by:

```
ADDRESS VALUE replyport

"CONTINUE"

ADDRESS
```

That's it for the calendar utility. Next month, I'll describe the set of ARexx programs that make up the rest of the reminder utility. ■

*Marvin Weinstein uses ARexx and REXX extensively in his work at the Stanford Linear Accelerator. Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (mweinstein).*

ON DISK

# Trouble-Free Data Sharing

By Eric Giguere

ON A MULTITASKING system, sooner or later one program will want to share its data with another program. To prevent confusion and to avoid corrupting that data, the programs will have to mediate access to the data. The Amiga offers four methods to accomplish this—message-based servers, lockfiles, forbidding, and semaphores.

## MESSAGE-BASED SERVERS

One way of sharing data or resources is to have one program act as a manager or server. Programs wanting access to the resources send a message to the server and wait for a reply. The reply message tells the waiting programs if their requests were honored and which resources they can access. When they are done, the programs return the resources to the manager for distribution to other programs. This can all be done fairly simply using techniques described in "Pass the Word: Interprocess Communication" (p. 35, June/July '91).

*"Remember to keep the time your program spends in a forbidden state to an absolute mininum."*

## LOCKFILES

A lockfile is a file that acts as a "key" to certain data or as an indicator of a system's state. When a program wishes to manipulate data, it checks if a lockfile exists. If one does, the program exits or waits until the lockfile is removed. Otherwise, the program creates the lockfile and continues processing. When done, the program removes the lockfile.

A lockfile may be (and usually is) an empty file. It's the existence of the file that is important. Lockfiles are easily implemented using standard AmigaDOS functions, such as Lock() and Open() or their equivalents in the compiler's I/O library.

Lockfiles are useful in situations where other methods are not practical, such as across a network or with AmigaDOS scripts. Lockfiles are also persistent—they survive system reboots (if not in RAM:). Normally this means your startup-sequence should remove any lockfiles that might be left around. A lockfile, however, can also be used for crash recovery if the program state is occasionally written to the file.

## FORBIDDING

The brute force method of data sharing is to use a pair of Exec functions called Forbid() and Permit(). Neither function should be used lightly, but sometimes they're the only safe way to access system lists.

Forbid() does just what its name implies: It forbids any other task from running until the forbidding task calls the Permit() function, at which point normal multitasking is reenabled. Forbid() allows a task almost exclusive control over the processor—only interrupts will be allowed to continue unimpeded.

If you call Wait() or similar functions after calling Forbid(), your task will be suspended and multitasking will be temporarily reenabled until your task becomes active again. Beware of most AmigaDOS input/output calls: they suspend your task while waiting for DOS requests to be processed.

You can nest calls to Forbid(), so multitasking is reenabled only when the corresponding number of Permit() calls are made. (When your task ends, however, multitasking is also restored even if you did not call Permit() the correct number of times.)

A typical use of Forbid() is as follows:

```
Forbid();

port = FindPort( "myport" );

if ( port == NULL ){
   port = CreatePort( "myport", 0 );
}

Permit();
```

Forbidding is necessary in this case to ensure that no other program creates a message port named "myport" between the calls to FindPort() and CreatePort(). You should also use this bracketing technique with the lockfiles to ensure no second program creates a lockfile between the time your program checks for the presence of a nonexistant lockfile and creates one of its own.

Remember to keep the time your program spends in a forbidden state to an absolute minimum. If you want to peruse system lists at your own convenience, copy the values you are interested in to memory you have allocated and examine the copies instead, because once you're out of the forbidden state those system lists can change on you at any time. Some system values can be examined within calls that are similar to but not as intrusive as Forbid() and Permit(). The Intuition functions LockIBase() and UnlockIBase() can be used when you want to examine Intuition's private structures, for example.

## SEMAPHORES

A semaphore is a specialized kind of forbidding, used when you wish to negotiate access to data without drastic

*Tired of snarled data transfer traffic? Police your programs with these routines.*

measures involving the whole system. Like a lockfile, a semaphore is a key that tasks can ask for to access shared data. When a task obtains the semaphore, it has exclusive control over the data. Other tasks wait in a queue until the first task releases the semaphore.

A semaphore is described by the Exec SignalSemaphore structure, defined in the exec/semaphores.h header file. To use a semaphore, a task must first allocate memory for the SignalSemaphore structure (setting the MEMF_PUBLIC flag when you call AllocMem() to obtain the memory is a good idea) and then call InitSemaphore() to initialize the semaphore. (It's extremely important that only one task create and initialize the semaphore!) At this point any task that knows the address of the semaphore structure can call ObtainSemaphore() whenever it needs to access shared data. The ObtainSemaphore() routine suspends a task until the semaphore is free again. A task calls ReleaseSemaphore() when finished with the semaphore.

The tricky part to using semaphores is passing the semaphore structure address to all the tasks that require it. Programs that split themselves into separate tasks using CreateTask() or AddTask() still share the same dataspace and so can use a global pointer to the semaphore (initialized before the task split). A shared library has its own dataspace and can also use this technique, with the semaphores initialized in the library start-up code.

For completely independent programs, Exec maintains a list of public semaphores that are accessible by name using the FindSemaphore() function. The code to add and release public semaphores—which must be done between Forbid()/Permit() calls because system data structures are being changed—is described in the *Amiga ROM Kernel Reference Manual: Libraries & Devices* (Addison-Wesley) and in the examples on the accompanying disk. (Note that calling ObtainSemaphore() after a Forbid() temporarily breaks the forbidden state, just as Wait() does.)

A second type of semaphore, message-based instead of signal-based, is also available under version 2.0 of the operating system: If a semaphore cannot be obtained immediately, the task is allowed to continue and will be notified via a message when the semaphore has been obtained. This method is more general than signal-based semaphores, but it is also more time-consuming. New signal-based semaphore functions are also available under 2.0.

An important point to note: Semaphores are implemented and supported by Exec, but they only work if every task uses them. A semaphore does not define the data that it is "protecting," and neither does Exec prevent tasks from modifying that data without first obtaining the relevant semaphore. Semaphores only work if each task cooperates.

## RUN ME ONCE ONLY

It's easy to have two or more copies of a program executing in memory simultaneously, but it's not always desirable. Programs that act as resource managers—servers—depend on having exclusive control over those resources. There can only be one boss in such cases.

The ARexx resident process is a prime case in point. It is responsible for launching ARexx programs and maintaining shared resources like the clip list and the library list. It doesn't make sense to run more than one copy of the resident process—it would be too confusing. If you use the rexxmast command when the resident process is already active, ARexx will politely inform you of this fact and refuse to start a second server.

So how do you go about ensuring that only a single instance is ever running? There are several approaches you can take. The first is to use a lockfile. A second approach is to use the Exec FindTask() function to search for a task by name, which is only useful when you create the tasks yourself using CreateTask() or something similar.

The simplest approach is to open a public message port with a fixed name—if you're writing a server program you'll have to do this anyway. When your program starts, it should first use FindPort() to see if the port already exists. If it does, your program exits immediately. Otherwise you immediately open the port via a call to CreatePort() and continue execution. The ARexx resident process uses this method by looking for a REXX message port when it starts up. Stop the resident process (using the rxc command) and run the norexx program on the accompanying disk. You won't be able to restart the ARexx resident process (using rexxmast) until you stop the program.

## FINAL NOTES

Data sharing has been a problem since the first multitasking operating system was developed, but using the methods we discussed you'll be able to avoid the corruption of shared data. The SAS- and Manx-compatible C programs in the accompanying disk's Giguere drawer are proof. ■

*Eric Giguere is the author of the* Amiga Programmer's Guide to ARexx *and a member of the Computer Systems Group at the University of Waterloo. Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (giguere) or Internet (giguere@csg.uwaterloo.ca).*

# TNT

*Technical News and Tools from the Amiga Community.*

Compiled by Linda Barrett Laflamme

## Take It Apart

Want to examine some code? To aid you, The Puzzle Factory has released version 5 of the **ReSource disassembler**. Written entirely in assembly language and promising to disassemble thousands of lines per second working from executable or binary files, memory, or disk blocks, Resource V5 runs on any 680x0 CPU and detects and uses an '020 or '030 chip if one is present. Output can be either traditional 68K syntax or Motorola's new M68000 Family assembly language syntax. The output is also compatible with the Macro68 assembler. With ReSource V5's over 900 menu functions you can go on to create real source code from the program's output. Most Amiga structure names can be accessed by pressing the appropriate key, and user-defined structures are also supported. New to version 5 is a hypertext on-line help facility that allows information on any function at the touch of a key. One caution: ReSource V5 requires at least one megabyte of RAM to run. For a price and other details, contact The Puzzle Factory Inc., PO Box 986, Veneta, OR 97487, 503/935-3709.

## Just in Case...

Once they have their source code, some programmers (but certainly not you) need help debugging it. Even if you're confident your code will run the first time through, keep in mind that INOVAtronics has acquired the publishing rights to the **MetaScope debugger**.

MetaScope is a multiwindow symbolic debugger with break points, trace, and assemble to memory options. Available directly from INOVAtronics, it retails for $99.95. Not that you'll ever need it, but in case your friend's code is buggy, the address to contact is INOVAtronics, 8499 Greenville Ave., Suite 209B, Dallas, TX 75231, 214/340-4991.

## Expand Up, Not Out

Interested in getting an A3000T, but can't afford one just yet? Consider the tower kit from INOVAtronics as an alternative. With the **HiQ A500 Tower** you can convert your A500 into a full blown tower Amiga. The HiQ gives your A500 four A2000 100-pin slots, one A500 coprocessor 86-pin slot, one video slot, two PC slots, key/lock security access, and a 220-watt power supply. All for $599.95 from INOVAtronics, 8499 Greenville Ave., Suite 209B, Dallas, TX 75231, 214/340-4991.

## Two Megs and Two OSes

If you're still searching for more hardware, consider what DKB Software is offering. For starters, there's the **MegAChip 2000** ($299.95). This card gives A2000 owners a second megabyte of valuable chip RAM, as well as extends the total amount of possible memory to 10 megabytes (that's two megs of chip and eight megs of fast). If you'd rather, you could add the **MultiStart II** ($99.95), a card that gives you the ability to have not just one but two Kickstarts in the same A500 or A2000 (Did someone say 1.3 and 2.0?) and to switch between the two via the keyboard. A1000 owners don't feel left out! The **KwikStart II** ($99.95) does the same thing for the A1000. There is an additional investment involved, however. Neither the MultiStart II nor the KwikStart II comes with ROMs. For complete details, contact DKB Software, 832 First, Milford, MI 48381, 313/685-2383. ►

# The AmigaWorld Tech Journal Disk

**ON DISK**

In addition to source and executables for the article examples, you'll find:

**Tons of Tools for Programmers**
**JPEG Compression Functions**
**Faster Blitter and 3-D Routines**

This nonbootable disk is divided into two main directories, *Articles* and *Applications*. Articles is organized into subdirectories containing source and executable for all routines and programs discussed in this issue's articles. Rather than condense article titles into cryptic icon names, we named the subdirectories after their associated authors. So, if you want the listing for "101 Methods of Bubble Sorting in BASIC," by Chuck Nicholas, just look for Nicholas, not 101MOBSIB. The remainder of the disk, Applications, is composed of directories containing various programs we thought you'd find helpful. Keep your copies of Arc, Lharc, and Zoo handy; space constraints may have forced us to compress a few files.

Unless otherwise noted in their documentation, the supplied files are freely distributable. Read the fine print carefully, and do not under any circumstances resell them. Do be polite and appreciative: Send the authors shareware contributions if they request it and you like their programs.

Before you rush to your Amiga and pop your disk in, make a copy and store the original in a safe place. Listings provided on-disk are a boon until the disk gets corrupted. Please take a minute now to save yourself hours of frustration later.

If your disk is defective, return to AmigaWorld Tech Journal Disk, Special Products, 80 Elm St., Peterborough, NH 03458 for a replacement.

# Get an F

Tired of Amiga Basic? Looking for a new programming environment? You should check out the latest version of F-Basic; version 4.0 is now available from Delphi Noetic Systems. **F-Basic 4.0** ($159.95) is an enhanced, compiled BASIC language system with extensive control structures, recursive subprograms, global variables, extended integer variables, fast nine-digit single-precision reals, double-precision reals, as well as text variables. Other key features include record structures, pointers, and high-level access to Amiga graphics, sound, menus, and the ROM Kernel. New to version 4 are an AREXX port, high-level gadgets, improved mouse events, separately compiled modules, automatic checksumming of programs, an improved editor, and Workbench icon arguments. Compatible with versions 1.3 and 2.0 of the OS, F-Basic 4.0 is available from Delphi Noetic Systems Inc., 2700 West Main St., PO Box 7722, Rapid City, SD 57709, 605/348-0791.

# Do-it-yourself Games

Feeling adventurous? Try your hand at **AEGIS Visionary**, a high-level authoring language for creating interactive graphic-adventure games. Sold complete with compiler and debugger, Visionary has a large number of commands for handling graphics and sound, as well a full set of text-manipulation routines. With Visionary you can build your own mouse-driven games in the tradition of Dungeon Master, text-only games reminiscent of Zork, and text and graphic games similar to Shadowgate. Visionary supports IFF graphics (including HAM), animations in IFF-Anim format, Aegis AudioMaster sequenced sounds, MED/MIDI music, and both NTSC and PAL displays.

The language has 70 programming commands, 19 math operators, 10 user-declared articles, 50 user-declared prepositions, definable function keys, virtual-page scrolling, speech output, and fast image blitting. The games you create can have up to 65,000 rooms with 32 attributes per room, 65,000 objects with each object having up to 32 attributes, 65,000 subroutines, 65,000 action blocks, 128,000 variables, 25 IFF graphic screen buffers, 25 IFF sound buffers, up to 50 on-screen gadgets, and stereo sound. (Whew!) Once you create your games you can compile and distribute them, even sell them! For further information, contact Oxxi Inc., PO Box 90309, Long Beach, CA 90809-0309, 213/427-1227.

# Look It Up

Need some new reading material? Consider perusing ***Program Design Techniques For The Amiga*** (£16.95, ISBN: 07457-0032-2) by Paul Overaa. New from the UK, the book covers a wide variety of programming techniques, including program design, structured programming, abstract data types, building "black box" modules, user interfaces, writing portable code, Copper-list programming, and an overview of ANSI C. All together there are over 400 pages of information with examples in ANSI C, BASIC, and 68K assembly. At this writing, Kuma Computers Ltd. does not have a U.S. distributor, so you must contact them directly at 12 Horseshoe Park, Pangbourne, Berkshire, England RG8 7JW, 011-44-734-844-335 (voice), 011-44-734-844-339 (fax).

# What's on Tap?

If you or your company has a hot new product on the way, tell us about it, and we'll tell the readers. Send your press releases and announcements to TNT, *The AmigaWorld Tech Journal*, 80 Elm St., Peterborough, NH 03458 or llaflamme on BIX. ■

## PROGRESSIVE PERIPHERALS & SOFTWARE'S

# 040

CALL FOR INCREDIBLE DEVELOPER PRICING!

LIMITED TIME OFFER. EXPIRES JAN. 31ST '92

### TIME TO MARKET IS EVERYTHING.

Compile time...test time...run time...it all adds up. As an Amiga developer, every minute of development time counts. Get your project out faster, and you're ahead of the competition. With the Progressive 040 accelerator, you'll spend less time and less money on projects, and get them out to market sooner. The Progressive 040/2000 and 3000 accelerators run at an average of four times the speed of a 25MHz 68030 accelerator, and nearly twice the speed of a 50MHz 68030. The Progressive 040 maximizes the power of your Amiga while minimizing computing time.

### IT'S YOUR OWN PERSONAL TIME MACHINE.

The Progressive 040 will get you there faster. The 040/2000 is the only 28MHz 040 accelerator available - the fastest on the market*. The 040/3000 is the first A3000 accelerator on the market. Both accelerators represent the state of the art in 040 architecture and engineering.

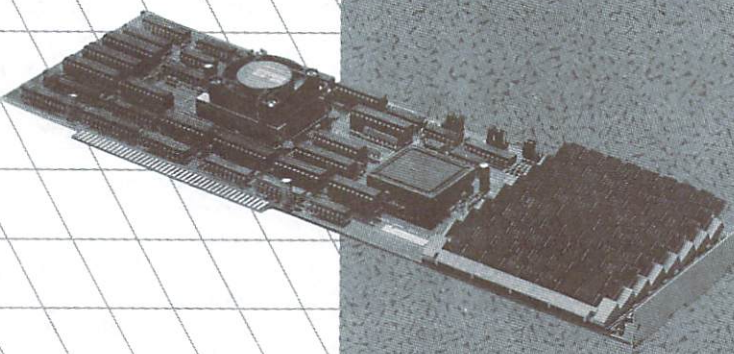
### NOW YOU CAN SAVE TIME AND MONEY.

Authorized Amiga developers qualify for incredible developer pricing - substantial savings over the Progressive 040's already low retail price. Call Progressive for details about this limited-time offer. Pick up the phone, and start saving time and money today!

*Based on several third party test comparisons performed August 1991

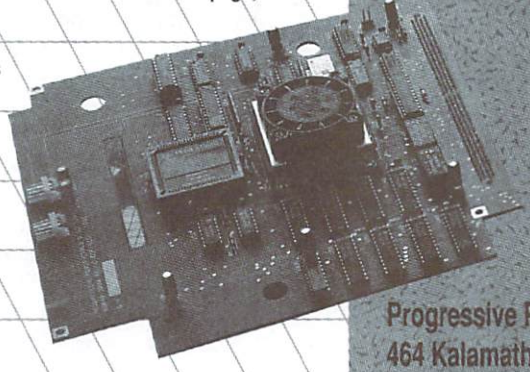
## 68040 ACCELERATORS FOR AMIGA 2000 AND 3000 SERIES COMPUTERS

- Motorola 68040 with built-in math coprocessor for speed and power
- Easy to install - plugs into Amiga 2000 or 3000 processor slot
- 19.2 MIPs (Million Instructions Per Second) performance at 25 MHz
- Processor cooled by whisper-quiet micro-fan for reliable performance
- Software compatible with all 68000 family processors
- 3.5 MFLOPS Double-Precision Floating Point Performance
- Separate 4K Data and 4K Instruction Caches
- Full support of 68040 "Copyback" mode for increased speed
- Relocates system vectors to 32-bit RAM for faster performance
- 040 Utilities and Floating Point Software Included
- Compatible with AmigaDOS 2.0, NTSC and PAL systems
- One Year Warranty



### PROGRESSIVE 040/2000

- 28MHz Asynchronous Operation
- Over 23 times the speed of a standard Amiga 2000
- AmigaDOS 1.3 and 2.0 Compatible - works with 1.3 or 2.0 ROM's
- Compatible with 3-D Professional, the Video Toaster®, Imagine®, and many other hardware peripherals and software
- Expandable to 4, 8, 16 or32 megabytes of 32-bit RAM using standard, low cost 1 MBx8 or 4MBx8 page, static column or nibble mode 80ns SIMM modules.
- Software switchable from 68040 to 68000 mode - no jumpers necessary
- Compatible with 16-bit memory cards and 'A' or 'B' series motherboards
- Designed for Upgradability to 33 MHz 68040 version



### PROGRESSIVE 040/3000

- 25MHz Synchronous Operation
- Full 25MHz performance on 16MHz A3000 systems!
- Directly accesses 32-bit memory on A3000 motherboard
- Over Four Times the Speed of a Standard 25MHz Amiga 3000
- Compatible with 16MHz, 25MHz, and "Tower" series computers
- Software switchable from 68040 to 68030 mode - no jumpers necessary
- Compatible with nearly all 2.0-compatible software and hardware
- Requires ROM-Resident AmigaDOS 2.0

Progressive Peripherals & Software
464 Kalamath St. • Denver, CO 80204
Phone (303) 825-4144 • Fax (303) 893-6938



Circle 10 on Reader Service card.

## Intuition: A Practical Amiga Programmer's Guide

*Not necessarily so…*

**By John Toebes**

WITH THE GOAL in mind of creating well-behaved Amiga programs, I am always on the lookout for good documentation and examples to add to my collection. At first glance, *Intuition: A Practical Amiga Programmers Guide (ISBN: 0-7457-0143-4)* by Mike Nelson appears to be just the book to help beginning Amiga progammers. Providing an introduction to Intuition functions on the Amiga, it covers the basics of Amiga programming, and offers an overview of the basic system functions.

The reasonably thorough introduction is followed by chapters such as "Screens and Windows," "Communications and Intuition," "Menus," "Gadgets," "Requesters," "Alerts," and "Preferences." In each of these sections, a complete sample program is given with the text interspersed between the sections of code. The text not only explains what the code is doing at each point, but also covers other topics related to the code.

For the true novice who is just becoming familiar with C, this can be a good learning environment (assuming that the reader doesn't attempt to type in the line numbers that appear beside each line of code and catches the places where the numbers flowed onto the code).This approach of mixing text and code is one of the things that makes the book attractive.

On the negative side, you must contend with the writer's continuous attempts at humor. In trying to explain what can be a very difficult subject to beginners, the author adopted a stand-up commedian monologue that at times is very distracting. This, combined with his self-professed lack of understanding of many of C's nuances, makes the book a poor choice for many beginning C programmers. To give you a taste, on page 27 he explains NULL with:

> *"The use of NULL for zero is a strange sort of inverted abbreviation, something I use because everyone else does, although I confess I'm not totally convinced about it! I guess it all boils down to readability in listings and, as we are using four characters instead of one, brevity cannot be the reason."*

### WE NEED THE WHOLE STORY

One important topic that I'd expect a European author to tackle is that some programs must run on both NTSC and PAL Amigas. Alas, you won't find that information here. Throughout the book, window sizes are hard-coded with 640/200 values, without hints on properly obtaining the values.

Moving on to menus is not much better. In an example on page 85, Nelson attempts to show how menu verify events work. While the example does show how they work to some degree, the author makes the mistake of attempting to do screen I/O through printf() while Intuition is waiting for the MENUVERIFY to complete. Following this lead, the beginner who is not aware of the limitations of MENUVERIFY will quite likely be faced with a locked-up Amiga and no good explanation.

Independent of this and other mistakes, the usage of menus is a very simplistic approach. No general techniques for modularizing menu creation are offered; the author constructs each menu by hand. As a result, the presented code fails to take into account the fact that users may selectively change their default menu font, causing the code to break quite spectacularly. It is worth noting that the code presented relies on the menu number to indicate function, making it difficult to update menus without having to also change program code.

With "Gadgets," the example program completely side-steps the issue of window border sizes and attempts to make hard coded values of 10 and 20 pixels act as a margin between the drawing surface and the edge of the window (to avoid overwriting the system gadgets). Given that there is already a border width in the Window structure (as shown in Appendix B on page 226), I wondered why the constants are used. Even when he tackles the extremely tricky subject of mutual-exclude gadgets, the author fails to mention the problems that every beginner will encounter when they don't use Image structures for the gadgets. (If you haven't seen it, they will come out in strange forms of reversed rendering after one cycle of clicking.)

In section 5.4, the author tells you that there are known problems with the handling of mouse up/down events in the code. (If they are known, why aren't they fixed?) It is this type of problem that the beginning programmer is likely to encounter (read that as "pull out his hair trying to understand the problem"). The solution of course is left for the reader to discover since it is only hinted at in the book.

The next 60 pages cover requesters and, in this particular case, we encounter yet-another-file-requester. Besides the flaw of this basic concept, there are a number of problems with the presented code. The author again fails to take into account the default system font and uses hard-coded screen limits. Going through the code,

I found several hidden gotchas. He uses hard-coded constants of 30 throughout the program both as a limitation on file size and as a place to store a flag that indicates the type of file. He fails to notice that a file name of exactly 30 characters will go right past the end of the array because of a known problem in the SAS/C strncp). This is one example where a well-placed data structure would make it easy on the beginner and actually get around the bug. The author's lack of thorough knowledge of C really shows in this example where he regularly uses: &FileList[i][0] instead of the more proper FileList[i] to reference a character array.

Another very important point that gets completely glossed over is dealing with proportional gadgets. As anyone who has had to hassle with the strange Amiga approach to ranging on the gadget knows, it is nearly impossible to get it exactly right. To get the right values, you have to either tweak the calculation until it seems to work (as the author admitted he did on page 193 for this example) or take advantage of one of the generalized (but quite small) subroutines that Commodore has published for this exact purpose.

The last chapter very briefly covers using alerts and even mentions the preferences routines. In general, you won't get much out of it.

The book has two large but mostly useless Appendixes. The list of Intuition functions in Appendix A suffers from a lack of formatting, making it nearly impossible to look up a function. Even more frustrating, once you find a function, the entry may not provide any more information than you already knew. For example, ClearPointer appears as:

> *"Function:*
> *ClearPointer(WIndow)(A0)*
> *Purpose:*
> *Zaps the arrow into oblivion, without showing any mercy for its feelings at all.*
> *Arguments:*
> *Simple needs, but devastating effects on arrow well being.*
> *Returns:*
> *Not even a piece of sausage."*

Appendix B is not much better. It includes a dump of all the Intuition-related structures. Unfortunately, all the comments from the original files have been stripped out and even the private portions of IntuitionBase are reproduced, leaving you with six pages of variable names and structures with no clue as to their real use.

## THE DARK CLOUD BEHIND THE (NOT-SO) SILVER LINING

Overall, this book has a significant number of technical flaws and limitations that should have been caught in the review process before press time. These take away from what little value the book had to offer. Among the many problems:

• Although the book is only recently published, it fails to mention some of the basics of the 2.0 operating system—details that have been available for quite some time. This single problem should cause you to think twice before using *Intuition: A Practical Amiga Programmer's Guide* as the basis of Amiga development. In fact, most of the book is made obsolete by techniques that 2.0 provides in the form of Boopsi, TagLists, and GadTools.

• Any programmer new to the Amiga should not live without prototypes and type-checking. None of the code examples, however, take advantage of this powerful feature. In fact, a number of useless casts would be eliminated through correct use of the supplied header files. As an example, look at page 13 where the author incorrectly complains about OpenWindow() returning an APTR (which it doesn't) and then suggests that you code the call with a cast such as:

```
MyWindow = (struct WINDOW *)OpenWindow(&MyNewWindow);
```

The author also spends time to complain about ANSI and the chip keyword, which in fact has been addressed by ANSI with the __chip keyword alternate. In another place he makes the incorrect observation that __aligned is not ANSI compliant.

• Index entries are replicated: In a very quick perusal, I found duplicate entries for File Gadgets, Planepick, STRINGCENTER, LONGINT, and Gadgets.

## IN CLOSING...

The book's real humor lies in the glossary, which sports entries such as:

> *"Aardvark: Usually the first word in such alphabetic lists, but of no real consequence as far as Intuition goes."*

Perhaps the same can be said about the book. For the beginner having trouble understanding a small aspect of code, the explanations in this book are sometimes helpful. If you want to learn how to properly program the Amiga, however, you are better off struggling elsewhere. ■

**Intuition: A Practical Amiga Programmer's Guide**
***Kuma Computers Ltd.***
12 Horseshoe Park
Pangbourne
Berkshire, England RG8 7JW
011-44-734-844-335 (voice)
011-44-734-844-339 (fax)
£16.95

ON DISK

# An Intuition Shortcut: EzLib

By Dominic Giampaolo

ONE OF INTUITION'S greatest features—its amazing flexibility—is also its downfall for simple projects. Using traditional methods, making a window with three buttons requires you to fill out 18 fields in the NewWindow structure and 45 fields for the three Boolean gadgets. Plus, you have to open intuition.library and graphics.library, error-check all your results, open the window, add the gadgets to the display, and then get to your main event loop. Of course, a good number of the 63 fields will be zeros or maintained by Intuition, but setting them up is still a tedious chore. What if you want a 3-D look for the gadgets? Tack on more code to create the necessary Border structures. If only you could request a simple Boolean gadget at an X,Y position with some specified text in it.

Consider using custom screens and fonts. For a screen, you can get away with a ten-item NewScreen structure. If you want a screen deeper than two bitplanes, however, get ready to write a considerable amount of code to allocate BitMaps and BitPlanes, again with full error-checking. You should be able to make a straightforward request—for example, a five-bitplane lo-res screen—and let the system handle the rest for you. To add custom fonts, you must open diskfont.library, fill out a TextAttr structure, open the font, and then close diskfont.library. When you just want a 12-point Helvetica font, it seems like an awful lot of work.

How about a standard yes/no requester? You can use an AutoRequest(), but don't forget about correctly filling in the three IntuiText structures, and properly checking widths and height. You just wanted a yes/no requester, not a hassle. Suppose you need to get a text string from the user. You could design your own requester or window with string, "OK," and "Cancel" gadgets, if you don't mind adding about 200 lines of code and data structure definitions. Wouldn't it be nice if there were a way to say "get a yes no answer" or "get a string from the user"?

These and other problems stem from the fact that Intuition offers enormous flexibility at the expense of easy use. For the most part, I don't care about 95% of the extras that Intuition offers. When was the last time you wanted to worry about the CheckMark field of a NewWindow structure or the BackPen field of an IntuiText structure? Sure, you can set these to custom values, but they make simple tasks needlessly annoying and cumbersome.

## HELP IS HERE

The result of my frustration is EzLib, a link-time library of 29 functions that make Intuition programming easier. With EzLib, you can whip up an Amiga program replete with screens, windows, and all kinds of gadgets in about ten minutes and a third as much source code as the standard method requires.

For example, consider opening a custom screen: You want a hi-res, interlaced, four-bitplane screen and don't care about opening libraries, allocating bitmaps and bitplanes, and error-checking all results. With the EzLib approach, all you do is:

```
screen = ezMakeScreen(HIRES|INTERLACED, 4);
if (screen == NULL)
   error_code_goes_here();
```

Ezlib handles everything—making sure that the Intuition and graphics libraries are opened, allocating bitmaps, and so on. You don't get tangled in code that's irrelevant to your problem, and you can more easily see what is happening in your program.

In writing EzLib, I traded off Intuition's often unused flexibility for ease of use. EzLib contains simple functions with only a few arguments that make it easy to code but are less flexible. I could have compensated by adding more arguments, but that would have created a different version of the problem I tried to avoid. The most arguments an EzLib function accepts is eight, and half of those specify dimensions (left edge, top edge, width, and height). The arguments to EzLib functions are the bare essentials (integers, pointers, and character strings) needed to specify the more common user interface objects.

Rest assured, the EzLib code is solid. None of the EzLib code generates Enforcer or Mungwall hits, and every function error-checks all arguments. You can pass NULLs where there should be window pointers, and EzLib will gracefully return a failure instead of crashing. All of the string and Boolean gadgets in EzLib have the proper 2.0 style 3-D look, even under 1.3. EzLib properly frees all allocated memory and does not unnecessarily fragment memory.

Despite what you might think, these functions do not add bloat to your code or slow it down. EzLib is less than 14K of solid, error-checked code. Because it is a link-time library, you only link in the code you use. Efficiency is less of a concern, as the routines contain mostly initializations and almost no loops. Remember the admonition of Kernighan and Plauger (paraphrased), "If you want to improve efficiency, don't diddle the code, change the algorithm." EzLib will not be a bottleneck in your programs.

EzLib functions come in three varieties (along with some miscellaneous other routines). *Make* functions create an interface object and display it, *create* routines are a level lower and generally just build the desired object, and *kill* routines

*Save your energies for coding your main algorithm; let this library of routines set up the GUI for you.*

remove the object and free its resources. The simplest calls with the least flexibility are the make-level routines. Create calls offer more flexibility at the expense of an added argument or two and the responsibility of ensuring that the object gets displayed. You can freely intermix calls to the make and create levels. EzLib functions only build the standard Intuition structures, so there are no new structures to learn. Once created, all objects can be manipulated and changed by your code except for the ExtData field of a custom screen (EzLib uses this privately).

Now, on to the details of using EzLib. Remember, this is simply an overview; the complete function-by-function documentation accompanies the library in the supplied disk's Giampaolo drawer.

## LIBRARIES

Let's start with the first thing an Amiga program does, opening libraries. We are all familiar with the common first lines of an Intuition program:

```
GfxBase = (struct GfxBase *)OpenLibrary("graphics.library", 33);
if (GfxBase == NULL)
   error_code_goes_here();
```

Of course, you must write almost exactly the same code for intuition.library and all other libraries your program requires. Consider EzLib's OpenLibs() call:

```
if (OpenLibs(GFX | INTUITION | REXX) == NULL)
   error_code_goes_here();
```

One call does it all. Graphics.library, intuition.library, and rexxsyslib.library are all opened with a single function call. Should one of the three fail to open, the others are closed down and NULL returned. Closing down libraries at the end of your program is equally simple with CloseLibs(). To close the three libraries opened above, call CloseLibs() with the same argument. Alternately, the argument you specify to CloseLibs() may be a subset of the libraries previously opened. For example, you could close rexxsyslib.library and leave graphics and Intuition open.

## WINDOWS

The next step for a programmer usually is to open a window. I won't repeat the tedious NewWindow structure, but suppose you need a simple window with the usual arrangement of a title bar, depth gadgets, a close gadget, and a sizing gadget. How does this grab you:

```
struct Window *window;

window = MakeWindow(NULL, 0,0, 300, 100);
if (window == NULL)
   error_code_goes_here();
```

The first argument (NULL) indicates you want the window on the Workbench, and the next four provide the window dimensions. Of course, we don't all want fully-dressed Intuition windows, so EzLib provides CreateWindow(). If you want a title bar along with VANILLAKEY messages, but no depth, sizing, or close gadgets, you would say:

```
struct Window *window;

window = CreateWindow(NULL, 0,0,300,100, WINDOWDRAG, VANIL-
LAKEY);
if (window == NULL)
   error_code_goes_here();
```

Again, the EzLib solution is simple and to the point. With CreateWindow(), you can completely specify all of the IDCMP and other flags you wish. Should you use MakeWindow() and decide that you need other IDCMP messages, you can always call Intuition's ModifyIDCMP().

## BOOLEAN GADGETS

Want to add gadgets to your window? Simply call MakeBoolGadget(). Let's add two Boolean gadgets to the above window, one that says "Tastes Great" and the other "Less Filling."

```
struct Window *win;     /* opened earlier */
struct Gadget *gadg1, *gadg2;

gadg1 = MakeBoolGadget(win, 50,50, "Tastes Great", 1);
gadg2 = MakeBoolGadget(win, 50,90, "Less Filling", 2);

if (gadg1 == NULL || gadg2 == NULL)
  {
     KillGadget(win, gadg1);     /* ok if gadg1 is null */
     KillGadget(win, gadg2);
  }
```

That's all there is to it. Instantly we have two three-dimensional buttons stacked in a window, one with a GadgetID of 1, the other with a GadgetID of 2. The text we provided is centered in the box, and it all looks nice. Should you need to know the specific dimensions of a gadget EzLib created, say for algorithmic layout of a display, you can look in the Gadget structure itself. If MakeBoolGadget() fails for any reason, it returns NULL. The KillGadget() routine is intelli- ►

gent enough to recognize a NULL argument, and simply returns if you happen to pass it one. This makes clean-up considerably simpler. KillGadget() also works on all three types of gadgets with the same exact call.

### STRING GADGETS

String gadgets require equally little effort. Suppose you would like a 200-pixel-long string entry field positioned at 75,50, with a default string of "Tanstaafl" and a GadgetID of 3. Type:

```
struct Window *win;     /* opened earlier */
struct Gadget *str_gadg;

str_gadg = MakeStringGadget(win, 75,50, 200, "Tanstaafl", 3);
if (str_gadg == NULL)
   error_code_goes_here();
```

The string gadget has a text buffer of 255 characters that you can examine when you receive a GADGETUP message for the gadget. If you wish to use the text contained in the StringInfo buffer, you must copy it to a private area. Other than that, there are no restrictions.

### PROPORTIONAL GADGETS

Proportional gadgets are perhaps the most useful interface objects and the hardest to program correctly. At best, they are murderously difficult without EzLib. First, applications should deal with their own numbers, not Intuition's; proportional gadgets require you to convert all your numbers into Intuition's and back. Second, real-time feedback from proportional gadgets is a feature of Intuition that no good programmer should disregard, as frustrating as it can be to implement. EzLib solves all of these problems simply and gracefully. The only type of proportional gadget EzLib doesn't let you create are joystick-style gadgets that move on both the X and Y axes. Consider a standard-fare vertical proportional gadget at 50,75, with a size of 25,100 and a GadgetID of 4:

```
struct Window *win;     /* opened earlier */
struct Gadget *prop_gadg;

prop_gadg = MakePropGadget(win, 50,75, 25,100, FREEVERT, 4);
if (prop_gadg == NULL)
   error_code_goes_here();
```

Assume your proportional gadget will have a range of 1 to 45 (as the top value will never be returned, however, you must pass 0 and 45), with the knob positioned at the top, and the knob size representing 10. The call is:

```
SetPropGadg(win, prop_gadg, 0, 10, 45);
```

To determine the value of the proportional gadget after the user modifies it, call:

```
val = GetPropValue(prop_gadg)
```

You will receive numbers in the range of 0 to 44 (in this case).

EzLib provides an easy method for doing interactive scroll bars, as well. This method uses callbacks, a concept borrowed from the XWindow system. Assuming you receive a GADGETDOWN message from Intuition and determine it is for the scroll-bar proportional gadget, you call RealtimeProp():

```
RealtimeProp(win, prop_gadg, some_function, data_ptr);
```

Win is your window pointer, prop_gadg is a pointer to the gadget the user is adjusting, some_function is the name of the function to be called when the value of the gadget changes, and data_ptr is a pointer to any arbitrary data to be passed to some_function. When the user releases the Prop gadget, control returns to your program.

This method works out very nicely. Let's assume you have a text editor with a scroll bar on the right. When the user clicks on the scroll bar, you call RealtimeProp() and, along with the window and gadget, give it the name of your scrolling function and a pointer to the text buffer (as the data pointer). Each time the gadget changes, your scroll function is called—trivial real-time scrolling. All your scrolling function needs to do is accept three arguments: the window pointer, the data pointer, and the new value of the proportional gadget (which is within your predefined limits). The only drawback is that the function RealtimeProp() takes over receiving IntuiMessages for your window. For most types of applications this is not a problem, but DISKINSERTED, NEWPREFS, and INTUITICKS messages will still arrive and be ignored until the user lets go of the gadget.

If you need more control in specifying your proportional gadgets—such as when designing one that takes advantage of the REL positioning and size modes of gadgets—you can use MakeVertProp() and MakeHorizProp(). Both functions accept flags and activation arguments that allow you to create proportional gadgets that always position or size themselves relative to the enclosing window. For total control you can use CreatePropGadget(), which allows you to specify all of a proportional gadget's relevant features.

### SIMPLIFIED, SIMPLIFIED, SIMPLIFIED

How often do you need a simple yes or no response from a user? "Erase this disk?," "Delete Selected Items?," and the ubiquitous "Really Quit?" are all simple yes/no questions. Intuition's AutoRequest() requires you to fill out three IntuiText structures and calculate dimensions for the requester. EzLib provides GetYN(). To ask the user "Really Quit Program?" with EzLib, all it takes is:

```
answer = GetYN(win, "Really Quit Program?");
if (answer == TRUE)
   quit_program();
```

Trivial to use and helpful in many instances, GetYN() is one of those handy little items you will wonder how you got along without.

Equally useful is GetString(). Obtaining a single text string, such as the user's name, can pose a problem in the midst of writing a much larger program. It distracts you from solving your real problem and increases the chances for bugs to show up. GetString() works simply and efficiently. All you do is:

```
UBYTE *string;

string = GetString(NULL, "A Title", "Default string");
if (string == NULL)
   got_no_string();
```

GetString() puts up a small window with a string gadget, a "Cancel" gadget, and an "OK" gadget. The first argument is the screen you want the window to appear on; NULL indicates the Workbench. The title string is the window title, and the default string (if not NULL) is placed in the string

gadget to start. When the user clicks OK or presses Return in the string gadget, you get a copy of the text in the string gadget. If the user clicks Cancel or the window's close gadget, you receive a NULL.

As the string returned to you is dynamically allocated, remember to FreeMem() any nonNULL value you receive. Because you must account for the NULL character, you must call FreeMem() with strlen(string) +1, not strlen(string).

Disk-based fonts can greatly enhance the appearance of any application; however, accessing them normally is a chore. EzLib's GetFont() simplifies matters. It accepts two arguments, the font name and its size. To use a 24-point Times font, the following code is sufficient:

```
struct Window *win;  /* assume opened earlier */
struct TextFont *txfont;

txfont = GetFont("times", 24);
if (txfont == NULL)
error_no_font();

SetFont(win->RPort, txfont);

/* intervening code */

CloseFont(txfont);
```

If the font cannot be opened, a NULL is returned. Otherwise, you receive a TextFont structure you can immediately SetFont() into your window. Of course, you can open and use multiple fonts simultaneously, switching among them via SetFont(). When finished, remember to call CloseFont() for each font you opened with GetFont().

Finally, EzLib has a few convenience functions: LacedWB() tells you if the Workbench is running in interlaced mode or not. MSG(string) prints the string on the console. Print(RastPort, string) prints text into a RastPort using the current font and location (set previously with Move()). Circle(RastPort, x,y, rad) draws a circle at x,y of radius rad. Line(RastPort, x1,y1, x2,y2) draws a line from x1,y1 to x2,y2. SetColor (screen, col_reg, col) sets the color register, col_reg, to the color col. Col is a number between 0 and 4095 (any of the Amiga's 4096 colors). Several predefined colors (BLACK, WHITE and such exotic ones as INDIGO) make the call even clearer to read.

Making short work of common Intuition programming dilemmas, EzLib frees you from wrestling with the complexity of GUI programming, giving you time to write code that solves problems . While EzLib does not offer all the flexibility of raw Intuition calls, the lost elements should not be missed in most applications. Besides, because the source is available, you can change or add things needed. ■

*Dominic Giampaolo is a student at Worcester Polytechnic Institute who has programmed Amigas for several years. Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

# Designing the User Interface: Proper Dragging

By David "Talin" Joiner

WHAT'S DRAGGING? Pulling a Workbench icon or a slider knob around the computer screen? Yes, but there's more to it than that. For the purposes of this article I'll introduce a more formal definition, because, as you'll see, there can be many kinds of dragging and we need guidelines that apply to all of them.

Dragging falls in the class of user actions called direct manipulation, where selecting and manipulating an object comprise a single action. Note, therefore, that dragging is neither Object–>Action nor Modal in nature.

I define a "drag" as the following sequence of user actions:

1. Move mouse pointer over a hit-zone.
2. Press left mouse button.
3. Move mouse while holding down left button (with appropriate visual or other feedback).
4. Release left button.

In addition to sliders and icons, many other items can be dragged—boxes, select regions, entities in a CAD program, and so forth. A few of these deserve special mention:

**Select Regions:** Many programs let you select a range of items (for example, icons or text) by dragging over the items. The items in the selection may be contiguous or discontiguous, depending on the program. (I'll cover selection in greater depth in another article.)

**Wires:** Some applications let you join components with "wire" links that are dragged from a connection point on one component to the connection point on the other. If the components are moved later, the wire is repositioned accordingly.

**Vertices:** Sometimes the object dragged is just a small feature of a larger object, such as a corner of a polygon or a data point on a connected graph. In such cases, numerous other features of the object may be affected by the drag.

**Groups:** A group of selected objects can be dragged just as easily as a single object, although there are some constraints on exactly how this is done.

## KINDS OF DRAGGING

Dragging is normally used to change the position, size, or shape of an object. For example, a window may be dragged to a new position or new size, and a polygon can be dragged to a new shape.

Other attributes can be affected by dragging as well: object attributes (such as the angle of an arc) and object relationships (such as the connecting wires mentioned earlier). Drawing, as in a paint program, is a special class of dragging.

## INITIATING A DRAG

Dragging is usually initiated by clicking the mouse button when the mouse pointer is positioned over a hit frame or sensitive area, such as a slider knob. Hit frames are usually rectangular, although they can be polygonal as well. Rarest is the "bit-test" hit frame, where the mouse position is tested against a single-bitplane mask.

When multiple objects are mixed together in a common area, you not only have to determine if a hit frame was clicked on, but which hit frame it was. I'll use the term "picking" to refer to this type of operation.

Some very small draggable objects, such as musical notes on a staff, require a hit frame larger than the actual object. Here we run into the problem of overlapping hit frames—suppose two musical notes are so close together that the pick coordinates are within both hit frames at once! There are several ways of resolving this "tie":

1. Pick the topmost object.
2. Pick the object with its center or most important feature closest to the mouse position.
3. Use a mask test against the actual glyph to break the tie.

However, each of these methods has problems for certain types of applications. A more sophisticated approach uses the "gravity technique," where the program scans each object and examines those that are near the mouse pointer. (One way of handling this is to give the mouse coordinates a width and height as well as an x,y location.) Then each feature of the object is evaluated based on its distance from the mouse pointer and its priority. (For example, the ends of a line segment have a higher priority than the line itself.)

These two factors are combined to produce a "gravitational attraction" for that feature of that object. Once the objects have all been examined, the object with the highest rating is picked. If no object has a high enough attraction, then the click fails and no object is chosen.

Note that the gravity technique also lets an object have "parts" that can be dragged separately. The pick function could return not only the address of the object, but a subcode representing the part that was picked.

A common type of part is a drag handle. For rectangular objects, drag handles are little squares that appear on the corners or sides of a selected object, and are used to resize the object. The drag handle on a corner of an object will drag that corner but not the opposite corner, while a drag handle on an edge will affect only that edge. For polygonal objects, there's a drag handle for each vertex that affects the position of that vertex and no other. If the user clicks on an object rather than its drag handle, then the whole object is moved rather than resized.

I think drag handles should appear only on objects that are

*Miss Manners would be proud of this guide to acceptable user-interface etiquette.*

in the selected state, and that resizing should be handled as a normal drag—that is, the object should not continue to resize after the mouse button is released. (Some programs terminate the resizing operation with the next mouse click, but I consider this inconsistent.)

Of concern is the fact that, in many programs, the data is represented internally in numerical units other than pixels. For example, in a drawing program the internal coordinates might be in millimeters rather than pixels. The question is, therefore, should the gravity comparison be handled in terms of millimeters rather than pixels. A further complication is the fact that some programs support a "zoom" feature that changes the ratio of the internal units (inches) to on-screen units (pixels). Picking behavior should be consistent in terms of pixels, in that, if you decide a pick can hit the corner of a rectangle from five pixels away, this should be true at all zoom levels. Other than that consideration, the unit used really doesn't matter from a user interface standpoint, although it may matter from an efficiency standpoint.

## WHAT HAPPENS DURING A DRAG?

Once the hit frame has been clicked on, the user has two choices: He can move the mouse, in which case the object should start to move, or he can immediately let go, in which case the object should return to its previous state and nothing else should happen.

This second case bears close attention. A number of programs make the object snap to the mouse position as soon as it has been clicked on. I consider this undesirable, except in the case of a "snap slider" (my term for a nonstandard slider, where the slider knob goes immediately to the mouse position, regardless of where in the container body the mouse click occurs). To avoid this, you must calculate a dragging offset. When the object is clicked on, your program should calculate the difference between the object's origin point and the pick point, and subtract this vector from the mouse coordinates each time a new mouse message is received. Thus, the object's position is affected only by relative movements of the mouse, and not by its absolute position.

Sometimes the user may want to release the object without dragging it, but his hand accidentally moves the mouse slightly and then he can't back out of the operation. (Note that this is seldom a problem on lo-res screens.) To avoid this situation, I like to make objects slightly "sticky," in that they don't start to move until the mouse has actually moved two or three pixels from the starting position. I normally do this by setting a special flag when the mouse is first clicked, and saving the mouse coordinates at that time. Each time I get a new mouse message, if the flag is set and the distance between the new point and the original point is less than three pixels, the object doesn't move. Otherwise, I reset the flag and calculate the dragging offset. If the flag is reset, I execute my normal object-dragging routine.

You can get even more sophisticated by letting the stickiness time-out—that is, if a drag starts and then just holds for a while, the stickiness automatically goes away.

## FAST FEEDBACK

Once actual dragging has started, it's important for the response to be snappy and to keep mouse messages from getting ahead of you. The following technique works well for me:

Define a flag that means "redraw the dragged object." (Actually, I have a similar flag for each of my program's rendering operations—scrolling, updating sliders, and so on. There's a separate set of flags for each open window.) Each time a new MOUSEMOVE message is received, store the mouse coordinates in a global variable and set the flag. However, don't update the object or redraw it, because Intuition messages usually come in batches. When the GetMsg() function returns a NULL (no more messages), the current batch is finished. At this time, most applications call Wait(), so they can put their task to sleep until the next batch of messages arrives. Instead of calling Wait(), however, you should first check the flag. If it's set, then update and redraw the object, and reset the flag. With this technique, mouse movement appears smooth and the application never lags behind.

Even more refinement is possible. Because internal coordinates can differ from on-screen coordinates, the mouse may move on the screen while the numerically-quantized internal coordinates stay the same. A similar effect can occur when the object is constrained (for example, you've dragged it up to the edge of the document). Don't redraw the object if its position didn't actually change, because it will flicker repeatedly, looking much like a strobe light as you move the mouse. I use the following technique to avoid this:

1. Subtract the dragging offset from the mouse coordinates.
2. Convert the mouse coordinates to the internal representation.
3. Apply miscellaneous constraints, such as grid snap, and clip the new coordinates to the document size.
4. Compare the new coordinates with the existing coordinates for the object. Note that, if you're resizing an object, the coordinates you may be comparing will be different than if you're just moving the object. For rectangular objects, it may be easiest to make a copy of all four coordinates (x, y, width, height) and have the mouse update some of them based on the type of dragging. ►

5. If the new coordinates are different from the old, erase the object at its current position, update the coordinates, and then redraw the object at its new position. You may also want to set a flag indicating that the whole document will need to be redrawn once the dragging is complete.

Even with all this, actually backsaving and redrawing the object each time is normally too slow. However, a COMPLEMENT-mode dashed outline of the object is a sufficient proxy for the real object, and is easily erased. For groups of objects, you can either draw the outline of each or an outline of the group as a whole.

Other types of feedback can occur during dragging. A number of applications have rulers at the edge of the drawing area, and, when the mouse moves, little indicators in the rulers move along with it, to indicate the exact position of either the object (when there is one) or the mouse (when no object is being dragged). Alternatively, a music program might have audible feedback as notes are dragged around.

*"The scrolling should occur at a constant speed, regardless of the mouse's movement."*

## SCROLL ADJUSTMENT

Many programs support documents larger than a window, allowing you to scroll around the document. When you drag an object to a part of the document that's not currently visible, the screen should scroll automatically to keep up.

Fortunately, you need not test the position of the object against the borders of the window to determine if a scroll is needed. Rather, this test should be made against the mouse pointer's real (not adjusted) position, which is simpler because the mouse pointer is just a single point in pixel coordinates.

For fast, pleasing responsiveness, the actual scrolling of the drawing area can be deferred until after the batch of Intuition messages has been processed, just like the object redrawing. I use two flags, actually—one to indicate that the screen should be scrolled and one to indicate that the border sliders need to be updated, because the sliders can cause scrolling.

The scrolling should occur at a constant speed, regardless of the mouse's movement. I use the INTUITICKS Intuition message for this. When I get an INTUITICK message, if an object is being dragged and the mouse pointer is outside the drawing area, I add or subtract an offset to the scroll position for that document (the numbers are clipped to the document size, of course) and then set the relevant flags. Scrolling then occurs automatically. (Also, the same flags can be used when dragging the border sliders.)

By the way, I prefer to use ClipBlit() rather than ScrollRaster() to scroll my display, and then build a clip region representing the edge of the display that needs to be updated. This is fairly easy, and avoids the annoying flash along the edge when ScrollRaster() clears the damage region to color zero. (Be careful about clip regions, however. I recommend that, any time you install a clip region in an Intuition window that contains Intuition gadgets, you have the layer locked during the time the clip region is installed. The reason is that when Intuition draws gadgets and window borders, it clones your RastPort but not your layer—clip regions can confuse Intuition greatly.)

## GRID SNAP AND OTHER QUANTIZATIONS

Many applications have a "grid snap" capability, which forces the position of the object to be quantized to a grid. This grid is nearly always specified in internal coordinates, and can be different in the x and y directions. Usually there are options to set the grid size and whether or not the grid is displayed in the drawing area.

One important consideration has to do with objects that are in the drawing area but not aligned with the current grid settings. Should the object maintain the relative "grid error" when being dragged, or should it align itself with the grid? In other words, is the grid a relative or absolute constraint? The answer depends on the application. For example, in a music program where notes are being dragged around, you'd probably want to preserve the grid error. When a musician plays a note slightly late, it's probably intentional. If you drag that note to a different measure, the note should move by exactly one measure and the same timing should be preserved. A separate tool can be used to align notes to the grid.

Grid error is preserved by maintaining a "grid offset" vector when dragging. The grid offset is calculated at the same time as the mouse offset. You'll also want to add one-half of your grid size (in pixels) to the mouse offset, so the object will snap to the nearest grid line instead of the nearest lesser grid line. Each time you quantize the object's position to the nearest grid line, you can then add in the grid offset so the error will be preserved.

Another type of grid snap is called "data snap" or "value snap." Data snap is used when dragging an attribute of an object that's not related to the x or y coordinate axis. For example, suppose you're dragging the radius of a circle. It makes little sense to quantize the radius to the grid square nearest the mouse; rather, quantize the actual "radius" value calculated from the unquantized mouse coordinates.

## FINISHING A DRAG

Releasing the mouse button signals the end of a drag operation. At this point, the application should be notified that a change has occurred and should take the appropriate action, such as redrawing or updating the screen. Note that, if the dragged object didn't actually move, this may be unnecessary; just erase any COMPLEMENT outlines that may be on the screen, and go back to waiting for the next mouse click.

For dragging rectangular objects, I like to create a clip region containing only the new and old positions of the object. I install that clip region, and then redraw the entire screen. As a result, only a small part of the screen is actually redrawn, and there's no delay for the screen to be erased and the individual objects to be redrawn.

The user should have the option to cancel a drag operation. I suggest that the right mouse button be used for this purpose, because that's the way Workbench 2.0 works.

I hope you haven't found this article too much of a "drag!" Next time we'll discuss the proper uses of fonts in your user interface. ■

*David "Talin" Joiner is the author of Faery Tale Adventure and Music-X, plus an artist, award-winning costume designer, and moderator of the user.interface topic of the Amiga.sw BIX conference. Contact him c/o* The AmigaWorld Tech Journal, *80 Elm St. Peterborough, NH 03458, or on BIX (talin).*

ON DISK

# MIDI Programming Demystified

*If you want your program to output to a MIDI device, here are the routines you need.*

By Darius Taghavy

DESPITE ITS MANY runtime libraries, the Amiga provides no operating-syestem-level MIDI routines. At the same time, programmers are urged not to program the hardware directly, but to use the Amiga's device interface. For MIDI this constitutes a predicament. The serial device, which generally does a formidable job, was not designed with MIDI in mind and adds too much overhead. The mere fact that you have to set a SERF_RAD_BOOGIE flag when opening serial.device to run MIDI does not exactly inspire confidence. In a nutshell, the serial device is completely inadequate to handle the high throughput of MIDI communications in professional applications.

MIDI is a rare instance when Amiga programmers simply have no choice but to address the Amiga's custom hardware directly. In this article, I will show you all the necessary steps to achieve OS-legal, low-overhead MIDI output on the Amiga's RS-232 port.

## CUSTOM CHIPS AND CUSTOM-CHIP REGISTERS

The Amiga's RS-232 serial port interface is controlled by the Paula chip, which also houses the floppy-disk controller, audio hardware, DMA interface, and more. Most importantly, this chip contains three registers of special interest to MIDI programmers. They are accessed by adding their respective offset number to the custom-chip base address ($dff000). The relevant registers are SERPER ($032 offset), SERDAT ($030 offset), and SERDATR ($018 offset).

The general rule when programming the Amiga is: Before you can access any of the custom-chip registers, you have to properly attain a lock on all the hardware resources to which those registers belong. This step is necessary because the Amiga is a multitasking system and you must prevent tasks from interfering with each other's I/O. Using the serial hardware is no exception! For this purpose the Amiga system utilizes *resources*. They provide a method of hardware arbitration and are used by the system devices themselves. You could envision the system as a three layer structure: The actual hardware is lowest layer, the resources are in the middle, and the devices are the highest layer.

To gain access to the serial hardware you must open misc.resource, which currently governs the serial and parallel hardware. Once successful, you need to allocate the SERIAL_BITS and SERIAL_PORT bits to own the RS-232 port. The following code implements these steps:

```
#include <exec/types.h>
#include <resources/misc.h>
#include <stdio.h>

/* missing OS 1.3 system protos and pragmas */
UBYTE *GetMiscResource(LONG,UBYTE *);
#pragma libcall MiscBase GetMiscResource 6 9002

struct Library *MiscBase=NULL;

void alloc_misc_resource(void)
{
UBYTE* user=NULL;

  if((MiscBase=(struct Library*)OpenResource(MISCNAME)))
    {
    if ((user=GetMiscResource(MR_SERIALPORT,"AWTJ Test"))==
    NULL)
      {
      printf("Successfully allocated MR_SERIALPORT\n");

      if(!(GetMiscResource(MR_SERIALBITS,"AWTJ Test"))==NULL)
        printf("Error: Can't get MR_SERIALBITS.\n");
      else
        printf("Successfully allocated MR_SERIALBITS.\n");
      }
    else
      {
      /* Note: MR_SERIALPORT allocation failed. At this point,
      you may check if user = "serial.device" and
      if true call the flush_device() function
      (explained below) and try again!
      */
      printf("Error: Can't get MR_SERIAL_PORT!\n");
      printf("%s owns it already!\n",user);
      }
    }
  else printf("Error: No misc resource ?$%#@^%$^#!\n");
}
```

The above routine (and all other C code supplied in the accompanying disk's Taghavy drawer) works under SAS 5.04, Manx 5.0, PAL, NTSC, and OS 2.0.

## GIVE IT UP, SERIAL DEVICE

A quirk in pre-2.0 OS code prevents you from getting a lock on the SERIAL_PORT bit if another task previously opened the serial device. This is because the serial device fails to deallocate this resource when serial.device is closed (its open count goes to zero). Don't worry, however, if you detect this case. Invoke the code below as flush_device("serial.device) to force serial.device to free up the resource by flushing it out ►

of memory (forcing the task to quit). If the return value is true, the function was successful. You may now reinvoke the allocation routine to obtain a lock on the hardware.

```
/*
   Function: Flush device from memory, causing it
             to free up all memory and
             resources that it was using.

   Inputs: UBYTE* device_name
   Outputs: BOOL (TRUE=Success, FALSE=Failure)
*/

#include <exec/types.h>
#include <exec/execbase.h>

extern   struct ExecBase *SysBase;

BOOL flush_device(UBYTE *device_name)
{
struct Device *device;
BOOL  removed=FALSE;

   Forbid(); /* need to lock access to DeviceList */
   device=(struct Device *)FindName(&SysBase->
   DeviceList,device_name);
   if(device)
      {
      RemDevice(device);
      removed = TRUE;
      }
   Permit();
   return (removed);
}
```

Commodore's *Amiga User Interface Style Guide* (Addison-Wesley) promotes a standard procedure for handling non-sharable resources that may be owned by a program and that may keep other applications from running: The programs should include options, perhaps as menu items, to allow allocation and deallocation of the resources. This permits multiple programs that use, in our case, the serial hardware (such as sequencers and editor librarians) to coexist in memory and hence, allows users to quickly switch between them without having to quit one to boot the other. The feature is already supported by some commercial Amiga applications. Get on the bandwagon.

## SETTING THE MIDI BAUD RATE

Once you "own" the serial port you can do with it whatever you want. The first preparation for sending out MIDI data is to set the correct serial baud rate. The MIDI baud rate is rather fast—31250 bits per second. To set it, you poke the correct value into the SERPER register. Unfortunately, this is not as straightforward as "move.w 31250,SERPER." Instead, you have to compute a "magic value," N, based on the formula:

```
N = ((clock_freq+(baudrate/2))/baudrate)–1
```

Before you curse Paula's chip designer, study the set_baudrate() function below. You will soon realize that this extra step is necessary to ensure that your software runs anywhere on the globe. All timing in the Amiga is based on a color clock interval that differs from country to country. The set_baudrate() function computes the correct magic value based on the baud rate you pass into it. The routine also returns whether the program runs on a PAL or a NTSC machine.

```
/*********************************************************
* Function: Set baud rate                               *
* ________________________________                      *
* Inputs: UWORD baudrate                                *
* Outputs: BOOL (TRUE=NTSC, FALSE=PAL)                  *
*********************************************************/
#include <exec/types.h>
#include <exec/execbase.h>

#define NTSCFREQ 3579545
#define PALFREQ 3546895

BOOL set_baudrate(UWORD baudrate)
{
extern struct ExecBase *SysBase;
UWORD *serper; /* PAULA serper register */
UWORD magicvalue;
BOOL  return_value=TRUE;

   serper = (WORD *)0xdff032; /* absolute address of PAULA +
   SERPER offset */

   /* account for PAL/NTSC difference */
   if((SysBase->VBlankFrequency)==(UBYTE)50)        /* PAL */
      {
      magicvalue=((PALFREQ+baudrate/2)/baudrate)–1;
      return_value=FALSE;
      }

   else magicvalue=((NTSCFREQ+baudrate/2)/baudrate)–1; /* NTSC */

   *serper=magicvalue;
   return(return_value);
}
```

The formula used in the function is derived as follows:

```
baudrate = 1/((N+1) * color_clock)
```

Solving for N yields:

```
(N+1) * color_clock = 1/baudrate
N * color_clock + color_clock = 1/baudrate
N * color_clock = 1/baudrate – color_clock
N = (1/baudrate – color_clock)/color_clock
N = ((1/baudrate)/color_clock) – 1
N = (1/(color_clock*baudrate)) – 1          ; 1/color_clock=clock_freq
N = (clock_freq/baudrate)–1
```

The next and final step forces a roundup of fractional values, because we're doing integer math:

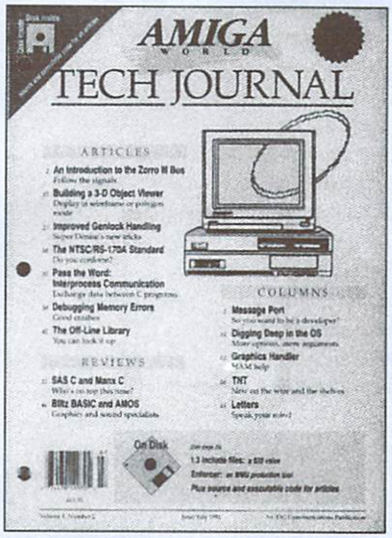
```
N = ((clock_freq+(baudrate/2))/baudrate) –1
```

Notes:

1. N is magic value and has to be a 15-bit number (bits 00–14).
2. A NTSC machine's color_clock is 279.365 nanoseconds, thus the NTSC color clock frequency is 1/279.365ns = 3579545 Hz.
3. A PAL machine's color_clock is 281.937 nanoseconds, thus the PAL color clock frequency is 1/281.937ns = 3546895 Hz.

As expected, the established baud rate is in effect for both reading from and writing to serial hardware registers. In case you wondered: There is no provision for reading SERPER to ►


April-May 1991 *Premiere*




June/July 1991


August/September 1991

# MISSING VALUABLE ISSUES?

## Send for your missing back issues and complete your

## AMIGA WORLD TECH JOURNAL

## library today!

Call toll free or mail this coupon today!

**1-800-343-0728**

Mail order to: AmigaWorld Tech Journal Back Issues
PO Box 802, 80 Elm Street
Peterborough, NH 03458
800-343-0728 / 603-924-0100

___ Apr/May 91 *Premiere* ___ Oct 91
___ Jun/Jul 91 ___ Nov/Dec 91
___ Aug/Sep 91

| | | |
|---|---|---|
| I have checked ___ back issues x $15.95 | = | $_____ |
| California orders add 6.25% sales tax | = | $_____ |
| Canadian orders add 7% GST | = | $_____ |
| Add postage/handling: | | |
| U.S. surface orders - $1 | | $_____ |
| Canadian surface - $2 | | $_____ |
| Canadian air mail - $3 | | $_____ |
| Foreign surface - $3 | | $_____ |
| Foreign air mail - $7 | | $_____ |
| Total Enclosed | | $_____ |

__ Check/money order enclosed
__ Charge my:
__ Mastercard __ Visa __ American Express __ Discover

Card # ____________________ Exp. ________

Signature ____________________

Name ____________________

Address ____________________

City, State, Zip ____________________

4CBIB2

## Listing 1: The Panic program ends stuck notes.

```
;-----------------------------------------------------------------------
; PANIC - written by Darius Taghavy — released into the public domain 1991
;-----------------------------------------------------------------------
;          Sends note # 0-127 with velocity 0 on all 16 MIDI channels
;          to get rid of those dreaded stuck notes!
;
;Demonstrates how to write assembler independant code by using Amiga standard
;assembler directives, and will hence assemble using any Amiga assembler that
;adheres to this standard.
;
;               assemble            link ;
;MANX:          as panic.asm        ln panic panic.o
;LATTICE/SAS:   asm panic.asm       blink panic.o
;-----------------------------------------------------------------------
;               SECTION CODE
XDEF            _panic              ;make available as 'C' function

                SECTION DATA
CUSTOM_CHIPS    equu                $dff000             ;custom chips base address
TBE             equu                5                   ;Transmitt buffer empty bit
SERDAT          equu                $030                ;serial data output register
SERDATR         equu                $018                ;ser.port data/status read

                SECTION CODE
_panic          movem.l             d2,-(a7)            ;push d2 on stack
                lea                 CUSTOM_CHIPS,a0     ;load custom chip base
                move.w              #15,d1              ;16 MIDI channels
channel_loop    move.w              #127,d2             ;128 MIDI notes
note_loop:
;-------------------status byte
                move.b              d1,d0               ;get MIDI channel
                andi.b              #$f,d0              ;clear status nibble
                or.b                #$90,d0             ;to or Note On status
                andi.w              #$ff,d0
                ori.w               #$100,d0            ;set stop bit
check_port_1    btst.b              #TBE,SERDATR(a0)    ;wait for free serial port
                beq                 check_port_1
                move.w              d0,SERDAT(a0)       ;send byte

;-------------------note number
                move.b              d2,d0               ;get note number
                andi.w              #$ff,d0
                ori.w               #$100,d0            ;set stop bit
check_port_2    btst.b              #TBE,SERDATR(a0)    ;wait for free serial port
                beq                 check_port_2
                move.w              d0,SERDAT(a0)       ;send byte

;-------------------velocity = 0
check_port_3    btst.b              #TBE,SERDATR(a0)    ;wait for free serial port
                beq                 check_port_3
                move.w              #$100,SERDAT(a0)    ;send 0 velocity

                dbra                d2,note_loop        ;loop 128 times
                dbra                d1,channel_loop     ;loop 16 times

                movem.l             (a7)+,d2            ;pop d2 of stack
                rts

                END
```

find out what the current baud rate is. It is strictly a write-only register.

### MIDI OUT

With the serial resources allocated and the MIDI baud rate set, you are ready to send MIDI data out the serial port.

The MIDI specification defines a MIDI byte to be eight data bits wide, flanked by one start and one stop bit. You have to send data out in this format to make sure the connected MIDI equipment respond as expected. The Paula chip, unlike most stand-alone UART (universal aysnchronous receiver transmitter) chips, does not have two separate registers—one to write data to and one to specify the data format. Instead, it has a single write-only, 16-bit register called SERDAT that is mapped into absolute address $dff030 (a $30 offset from the custom chip base, $dff000).

*"The MIDI specification defines a MIDI byte as eight data bits wide, flanked by one start bit and one stop bit."*

Into SERDAT you must write the actual data bits in a form that also defines the desired format of the data packet. As a convenience, the hardware automatically introduces every serial data message with a start bit, so you don't have to worry about that part. The machine, however, is capable of sending any combination between one data bit followed by 15 stop bits and 15 data bits followed by one stop bit; so you have to be explicit about what you want. Fortunately, the rules are quite simple: Data is shifted out until all set bits are sent. Hence, for the MIDI byte format of eight data bits and one stop bit, you load the lower byte of your 16-bit register with the MIDI data, clear the upper byte, and write a 1 into bit position 8 (the upper byte's least-significant bit).

Data transmission begins immediately after the data word has been written to the SERDAT register, provided the TBE (transmit buffer empty) bit in the SERDATR register is set. You validate this condition by testing the TBE bit and looping until the condition is true. When this condition is satisfied, you can write your data to SERPER to be shifted out the RS-232 port.

The following assembly code segment demonstrates these principles and sends a MIDI byte out the RS-232 port, assuming data is in the least significant byte of register d1. (High-level programmers please don't be put off by the assembly language. The examples are straightforward and will assemble and link with using any Amiga assembler/linker, because they use assembler-independent directives, which you should practice for portability.)

```
    lea     CUSTOM_CHIPS,a0     ; Load custom chip base address.
    andi.w  #$ff,d1             ; Clear upper byte; set data format
    ori.w   #$100,d1            ; to one stop bit and 8 data bits.
1$  btst.b  #TBE,SERDATR(a0)    ; Is serial transmit register ready?
    beq     1$                  ; If no, try again.
    move.w  d1,SERDAT(a0)       ; If yes, send out MIDI byte!
```

(Note: Waiting on the TBE bit constitutes polled I/O. This techniquee wastes time (320µs) in a busy wait loop and should not be employed within an interrupt routine. Commercial applications should place outbound data in a queue that is primed by an interrup routine tied to the TBE vector.)

With only the above concepts you can develop a small but useful utility to silence the all-too-common stuck-note phenomenon. The problem occurs because MIDI uses two distinct messages to define a musical note: a NOTE ON event and a NOTE OFF event. The elapsed time between these two events defines the duration of the note. Sometimes a NOTE OFF event gets lost, resulting in an infinitely sustaining note. The creators of the MIDI specification defined an ALL NOTES OFF event, but not all tone generators respond to it. Assuming the physical connection is not broken, the only foolproof way to silence the offending note, is to send a NOTE OFF event for every pitch on every channel. This "panic" feature is present on most professional sequencers, master keyboards, and MIDI switchers, as well as on some synthesizers. See Listing 1 for a tightly written assembly implementation.

### DEALLOCATE AND DONE

When you're ready to quit your application, or if a user requests you to give up the serial resources temporarily, you must properly deallocate the MR_SERIALBITS and MR_SERIALPORT bits you previously locked. It's as simple as making the calls:

```
FreeMiscResource(MR_SERIALBITS);
FreeMiscResource(MR_SERIALPORT);
```

### YOUR TURN TO MAKE NOISE

You now have all the necessary routines for system-legal MIDI output, let's put them to the test on your Amiga. Hook up your MIDI interface's line-out to a MIDI input on a tone generator (preferably a multi-timbtral unit). Then, from a command line (Shell or CLI) run the following programs from the Taghavy drawer in order:

```
alloc_rs232   /* allocates serial resources */
flush         /* if above program fails due to serial.device */
setbaud       /* sets MIDI baudrate */
stuck         /* generates a stuck note on every MIDI channel */
panic         /* quiets all stuck notes */
free_rs232    /* de-allocates serial resources */
```

With your new knowledge, there is nothing to keep you from adding low-level, low-overhead MIDI out support to your applications. As I demonstrated, MIDI output is simple; anybody can do it!

MIDI input, on the other hand, gets a little more involved as it requires writing two interrupt handlers (a receive-buffer-full handler and a timer interrupt routine) and possibly an extra task to be signalled by the timer interrupt. It also requires you to use a real-time data-buffering scheme with 320µs granularity and an associated data structure. Furthermore, it also necessitates an explanation of CIA hardware timer allocation and initialization. Watch for future articles on these topics. ■

*Darius Taghavy, formerly a Commodore software engineer, now works on product development for Ensoniq Corporation. He spends much of his dwindling spare time as a freelance (song) writer, computer/audio/MIDI consultant, and occasionally, beta tester. Contact him c/o* The AmigaWorld Tech Journal, *80 Elm St. Peterborough, NH 03458, or on usenet (darius@cbmvax).*

ON DISK

# A Developer's Guide To PostScript—Part I

By Tomas Rokicki

BECAUSE POSTSCRIPT IS the standard for portable graphics, and PostScript fonts comprise the largest and most widely used font library in the world, Amiga programs need to support the inclusion of PostScript graphics and fonts. Doing this correctly is not trivial; subtle interactions between the including application and the included graphics can cause fatal PostScript errors. Even if some of the rules can be bent on the Amiga, it is important to get them all correct anyway, because Amiga graphics may be imported to other machines for printing, or PostScript graphics from other machines may be moved to the Amiga.

In addition, PostScript's device and resolution independence provides one aspect of a solution to retargetable graphics on the Amiga. (The final result on different devices differs only according to the capabilities of the devices, because the image is described, rather than given bit by bit.) Adding a back end to a PostScript interpreter to drive another graphics card is generally very easy to do; the same cannot be said of the Amiga graphics.library. Almost all workstations sold these days support PostScript in their windowing system, either directly or through a supplied interpreter.

This two-part article presents a guide to PostScript for Amiga developers. Here in Part I, I will describe the language and the interpreter and give some details on rendering. In Part II, I will talk about the required structured comments that allow a PostScript file to be printed on a variety of systems and to be included as a graphic in other applications. I will also give an example of "proper" PostScript and some general guidelines for application programmers next time.

## THE PROGRAMMING LANGUAGE

What is generally referred to as the PostScript language can be split into two parts—the imaging model and primitives that are concerned with rendering graphics, and such programming language features as variables, dictionaries, and control structures. Before I introduce the imaging model, let's examine PostScript as a programming language.

PostScript resembles the Forth language, but the differences are almost as great as the similarities. Like Forth, PostScript is stack-based. The primary stack is the operand stack; values are passed to procedures and primitives by first placing them on the operand stack, and then executing the appropriate procedure. PostScript contains a full set of stack operations for removing, copying, and adding new values to the operand stack. In addition, numbers, string, or other literal values are not passive values; rather, executing a number has the action of placing its value onto the operand stack. This organization means that operands must precede the operator. This is generally referred to as reverse polish notation (RPN). To subtract 9 from 20 in PostScript, instead of 20–9, you type:

```
20 9 sub
```

The sub primitive is defined to remove the top (or most recent) two values on the stack, make sure they are numbers, take their difference, and place the result on the stack. Assuming the stack is initially empty, the operations and the stack results are as follows:

| Operation | Stack contents afterwards |
|---|---|
| 20 | 20 |
| 9 | 20 9 |
| sub | 11 |

(All stack descriptions in this article are described with the top of the stack at the right. Thus, values are always added to or removed from the top of the stack, or, in our illustrations, the right side of the stack.)

## THE SCANNER

The PostScript interpreter must be able to separate the incoming data stream into operations and values, ignore any comments, and determine the type of each operation or value. These operations are covered by the syntax of the language and are handled by the PostScript scanner. Like Forth, PostScript has an extremely simple syntax.

The minimum syntactic element is called a token. In PostScript, a token can be a name, a literal name, a number, or a string. Any occurrence of a percent character (%) outside of a string starts a comment; comments continue to the end of the line and are ignored by the interpreter.

Like most languages, PostScript is free-form. Spaces, tabs, and newline are treated as white space; they serve no purpose other than separating tokens. Other than white space, any of the characters below will terminate a token being scanned. They are not included in that token.

```
()<>[]{}/
```

White space and any of the above characters can occur inside strings (other than ")", which is used to terminate a string), although it is recommended that tabs and newlines be included in strings only by using the appropriate escape sequences. (To include a ")" inside a string, precede it by a backslash; details on this and other escape sequences can be found in the *PostScript Language Reference Manual*, also called the Red Book.) In addition, the characters:

```
[]{}
```

*Everything you need to know to program PostScript.*

*This time: the language, the interpreter, rendering details.*

are always scanned as single-character tokens, no matter what follows them.

Once a sequence of characters has been assembled into a token, the interpreter must determine the token's type. If the token is a syntactically valid integer, real, or radix integer, it is converted into a literal number token. If the token's first character is an ( or <, the token is converted to a string literal. In the former case, escape sequences are scanned and converted to the appropriate character; in the latter, the characters are interpreted as a hexadecimal string literal and converted to binary.

All other tokens are interpreted as names. (Even a token such as 123-456 is a name since it is not a valid number or string.) If the first character of a name is a /, then the token is a literal name. (More about this in a bit.)

Thus, the PostScript fragment:

```
123/3-99{add<6565>6565 -/-
```

is broken into a numeric literal of 123, a literal name of 3-99, an open brace token, an add token, a string with the value AA, a numeric literal of 6565, a name of -, and a literal name of -.

## EXECUTION

Normally, after each token is scanned, it is executed. Literal tokens are pushed onto the top of the stack. Nonliteral names are looked up and their associated value executed. The association between a name and a value is created by the def operator. Thus:

```
/n 3 def
```

associates the value 3 with the name n. When executed, a subsequent occurrence of n pushes the literal value 3 onto the stack. Note how definitions are handled in exactly the same way as any other operator; the top element of the stack is treated as a value and the second stack element is used as a "key;" the two values are removed from the stack and the interpreter remembers the association.

Literal arrays are created with the [ and ] operators. The [ operator (also called mark) simply places a special "mark" object on the stack. The ] operator counts the number of objects on the stack up to the first mark, allocates memory to hold that number of objects, copies those objects into the new memory while removing the objects from the stack, pops the mark from the stack and finally places a pointer to the array onto the stack. Thus, the sequence:

```
[ 1 2 3 /add 9 ]
```

pushes an array of length 5 onto the stack, while:

```
[ 1 2 3 add 9 ]
```

pushes an array of length 3 onto the stack (consisting of the elements 1, 5, and 9); the add is executed because it is not given as a literal.

Procedure bodies are just executable arrays, typically filled with operators and literals. They are created with the { and } operators, in a manner very different from the way arrays are constructed. The [ and ] are normal operators, while { and } change the scanner's operation. After the initial {, the scanner goes into deferred-execution mode, so objects are scanned and saved, rather than scanned and executed. Any nested { or } operators are recursively handled according to these rules, but nothing else is executed. When the matching } is encountered, the elements that have been saved are collected into an array and placed on the stack, as with ], except that the array is marked as executable. To execute an array so marked, the array elements themselves are executed in order.

To define add in terms of sub, for instance, you would type:

```
/add { 0 exch sub sub } def
```

When you later use add, the executable array is used as the operation. A 0 would be placed on the stack, and then the exch operator would be executed, which exchanges the top two elements of the stack. If the stack was originally 3 4, the stack would become 3 0 4. Then, a subtraction would be executed, subtracting 4 from 0, resulting in a stack of 3 –4, and a final subtraction would leave a final result of 7 on the stack.

## DICTIONARIES AND BINDINGS

The associations between keys and values (henceforth called bindings) are stored in a data structure known as a dictionary. Dictionaries can be allocated with the dict operator. Executing 10 dict, for example, leaves a dictionary on the operand stack that is large enough to handle 10 bindings.

Note that while keys are normally literal names, in truth they can be any object, including numbers, arrays, and even dictionaries. Not allowing all objects to be used as keys is a common bug in early releases of some PostScript interpreters.

In addition to the operand stack, PostScript also uses a save stack, an execution stack, and a dictionary stack. Mostly invisible to the user, the execution stack is used much like a subroutine stack in other languages. The save stack we will discuss later. The dictionary stack is scanned from top to bottom when a key is looked up, so recent dictionaries can override previous dictionaries. Definitions created by def are always added to the dictionary at the top of the dictionary stack. The begin operator takes a dictionary on the operand ►

stack and adds it to the top of the dictionary stack; the end operator removes the topmost entry of the dictionary stack.

One problem with PostScript is that bindings are dynamic—there is no efficient way to get named variables local to a specific procedure. If one subroutine uses a variable called i as a local index variable, and it calls another subroutine that also uses an i variable, the second subroutine will clobber the value used in the original subroutine.

As a solution to this, each subroutine can allocate a dictionary for its local variables. Alternately, the save/restore mechanism can be used to provide local contexts. Neither approach works well in practice. Allocating a new dictionary or a new save context for each subroutine can be slow. There are also typically small static limits on the sizes of the dictionary stack and save stack (20 and 15, respectively) that would severely limit the subroutine depth possible. Instead, all procedures in a section of PostScript code are generally treated as globals and stored in a single dictionary. The programmer simply needs to be sure to use unique names.

The bind operator in PostScript is rather a misfeature, and some interpreters feature a default autobind mode that is even worse. Taking a procedure as its single argument, bind replaces all names in the given procedure that currently bind to operators with pointers to the operators. Normally, you use bind immediately before def in a procedure definition. The intent is to bypass some of the name lookup overhead during execution by looking up the names during definition instead.

Unfortunately, the state of the dictionary stack, and especially the definitions in the current dictionary, tend to be different at the time of definition and the time of execution. Using bind can easily catch definitions in dictionaries other than the ones that were intended or might preclude overriding some other operators. To help address these problems, all local variables to a dictionary should be defined in the dictionary before you define procedures that use those variables. For example, the following sequence is correct:

```
/MyAppDict 10 dict def MyAppDict begin
  /x 0 def /y 0 def % provide names for possible binds
  /draw-line { /x exch def /y exch def ... } def
end
```

Omitting the definitions of x and y will usually work, but will fail occasionally because an including dictionary might already have x or y defined. Usually is not good enough, so the second line must be included.

## CONTROL OPERATORS

PostScript also contains a full complement of control operations that are stack based, rather than syntax based. For instance, if takes two parameters on the stack—a Boolean value and a procedure. If the Boolean evaluates to true, then the procedure is executed. Here is an example:

```
isLandscape { rotateCoordinates } if
```

Here, isLandscape evaluates to either true or false. If it is true, then rotateCoordinates is executed. Note you must use braces around rotateCoordinates to effectively push a literal procedure onto the stack; if you omit the braces, then rotateCoordinates is executed before the interpreter even sees the if.

PostScript also contains such control operators as ifelse, loop, repeat, for, and exit. For further details, consult the *PostScript Language Reference Manual.*

## IMAGING MODEL

Now, let's examine the basics of the PostScript imaging model. This imaging model is based on a single concept—filling an arbitrary path with opaque paint. All PostScript imaging operations eventually are defined in terms of this single operation. Thus, PostScript does not have drawing modes such as exclusive or, or, and, or other Boolean combinations of source and destination. The paint is always opaque and covers whatever is underneath.

All coordinates in PostScript are treated as floating-point numbers, even if the numbers used to specify them are integers. Coordinates are given in pairs, because PostScript graphics are always two-dimensional. The initial coordinate system in PostScript has 72 units to the inch, with the origin at the lower left of the page. X increases to the right, and y increases up the page.

## PATHS, FILLS, AND STROKES

A "path" in PostScript is a possibly unconnected sequence of lines or curves. When the path is filled, each group of connected lines or curves (called a subpath) is automatically "closed" with a straight line, if necessary, and then filled.

You can use several operators to create a path. The newpath operation clears the current path and sets the current point to undefined. Moveto takes an x and y coordinate and moves the current point to the supplied position, without adding a line or curve to the current path. The lineto operation takes two coordinates and adds a line from the current point to the specified point. The closepath function connects the current point to the start of the current subpath (typically the coordinate of the most recent moveto command.) Finally, fill paints the area defined by the path with the current color. The current color is a component of the graphics state that you set with the setgray command; 0 is black and 1 is white. The fill operation automatically closes any open subpaths before painting. The fill operation consumes the current path; after the fill, there is no path or current point defined.

If a path intersects itself, as in Figure 1, the fill operation is a bit more complicated. A winding number for each area is calculated by projecting a vector out of each area towards infinity, in any direction. The winding number is initialized to zero. As the vector is chased to infinity, if a segment of the path crosses this vector from right to left, the winding number is incremented. If a segment crosses the path from left to right, the winding number is decremented. Figure 1 lists the winding numbers for each region in the leftmost graphic.

With the standard fill, all areas that have a winding number that is not zero are painted; this is referred to as the nonzero winding rule and is illustrated by Figure 1's middle graphic. PostScript also supplies the eofill operator, which fills only areas with odd winding numbers (shown in the rightmost graphic of Figure 1).

A path may also be stroked. The stroke operation uses the current path as the center line of lines to be drawn in the current color. When a path is stroked, subpaths are not closed.

Various parameters in the current graphics state are used when lines are drawn. The current line width can be set with setlinewidth and is in the same units as the coordinate system. The default initial value is 1, which initially corresponds to 1/72 of an inch.

Additional graphics state parameters are also necessary to draw lines, and you can change these with setlinecap and setlinejoin. These parameters indicate how the ends of lines and

the join between two lines meeting at an angle are drawn. The line cap is used at the two ends of a subpath, even if the two ends of the subpath are at the same point. If the subpath is intended to be closed, you should use closepath at the end of the subpath to render it properly.

Inside the PostScript interpreter, stroke operations are converted to fill operations before rendering. Lines, caps, and joins are converted to polygons, and then the polygons are filled. The strokepath operation gives you access to this conversion process; it replaces the current path with a path that would be painted by stroke. Nominally, strokepath fill is equivalent to stroke, except that stroke is usually much more efficient. Figure 2 uses strokepath to illustrate how this is handled inside the interpreter, using the default miter join. Note that different interpreters might use different sets of polygons, but the total area covered will be the same.

Like lineto, curveto extends a path. This operation draws a Bezier spline, which is defined by a start point, an end point, and two control points. The Bezier spline begins at the start point in the direction of the first control point and terminates at the end point coming from the direction of the second control point. The curveto operation uses the current point as the start point and takes as parameters the first control point, the second control point, and the end point, respectively.

Figure 3 illustrates some Bezier splines. A useful property of a Bezier spline is that it always lies completely within the polygon defined by the start, end, and control points. (For more on Bezier curves, see "Better Bezier Curves," p. 62.)

Inside of the PostScript interpreter, splines are rendered as a sequence of lines. Of course, no curve can be perfectly drawn by lines; more lines yields a slower but more accurate rendering. You control the rendering accuracy via setflat. The setflat operation specifies, in output device pixels, the maximum deviation from the true curve that the approximation is allowed to have. If you increase the parameter, fewer line segments are used to render curves; if you decrease it, more are required.

PostScript can also render arcs—portions of circles. The arc, arcto, and arcn operations add arcs to the current path. (Refer to the Red Book for more information on how to use them.) Arcs are internally reduced to Bezier splines—a single Bezier spline can represent a 12-inch-radius quarter circle with an accuracy of better than 1/300 of an inch.

You can also draw dashed lines; these are enabled with the setdash operation. PostScript reduces dashed lines to a sequence of normal, solid lines before rendering, and then renders each of these individual lines.

## COORDINATE TRANSFORMATIONS

One of PostScript's nicest features is its ability to perform general coordinate transformations. The default coordinate system described earlier can be rotated, scaled, and skewed arbitrarily, and all graphics state parameters are automatically transformed. The rotate and scale primitives change the

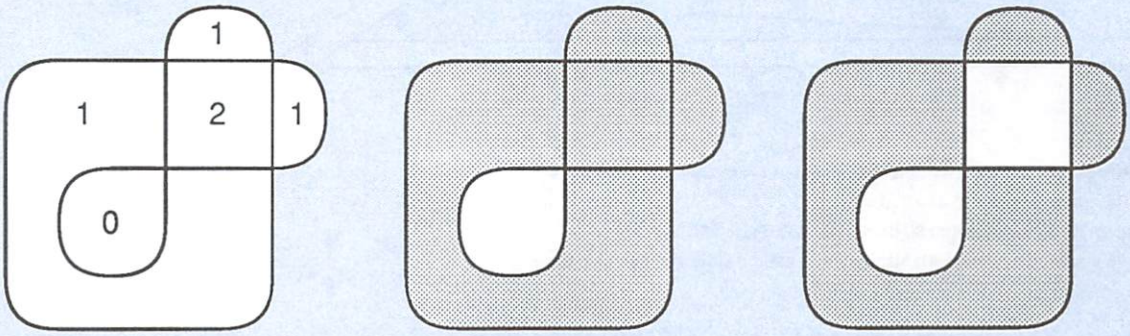



Figure 1: A self-intersecting path with winding numbers drawn with fill and eofill.

```
%! PostScript, but most structured comments removed for brevity.
%%BoundingBox: 70 70 494 194
/Helvetica findfont 12 scalefont setfont
/x1 0 def /y1 0 def /x2 0 def /y2 0 def /p 20 def /n -20 def
/C { /y2 exch def /x2 exch def /y1 exch def /x1 exch def
  x1 .7 mul dup y1 .7 mul dup 3 2 roll x2 .3 mul add exch
  y2 .3 mul add x2 y2 rcurveto } def
/R { rlineto } def
/fig { newpath 0 p moveto 0 p 3 mul R 0 p p p C p 4 mul 0 R p 0 p n C
  0 n n n C n 3 mul 0 R n 0 n n C 0 n p n C p 0 p p C 0 p 3 mul R
  0 p p p C p 0 p n C 0 n 4 mul R 0 n n n C n 3 mul 0 R n 0 n p C
  closepath } def
/S { p mul exch p mul exch moveto -4 -4 rmoveto show } def
gsave 72 72 translate fig stroke (1) 1.5 4 S (2) 4 4 S (1) 5.5 4 S
  (1) 4 5.5 S (0) 2 2 S grestore
gsave 222 72 translate fig gsave 0.9 setgray fill grestore stroke grestore
gsave 372 72 translate fig gsave 0.9 setgray eofill grestore stroke grestore
showpage
```

coordinate system by rotating it around the origin or scaling it independently in the x and y directions; the transform operation generalizes this by applying an arbitrary linear transformation to the current coordinate system.

Handle this feature with care. For instance, if you wanted to stroke a set of rectangles, you could try:

```
/rectangle { % usage: w h rectangle –
        % Strokes a rectangle w wide and h high
   gsave
      scale % use scale by width and height to set the size
         % then draw a unit square and stroke it
      0 1 rlineto 1 0 rlineto
      0 –1 rlineto closepath stroke
   grestore
} def
```

(We will discuss gsave and grestore later.) Unfortunately, the above will not have the desired effect, because the line width is scaled with everything else. Therefore, the width of the lines drawn will vary according to the size of the rectangle. The solution is to use the width and height as parameters to the rlineto commands:

```
/rectangle { % usage: w h rectangle -
      % Strokes a rectangle `w' wide and `h' high
   /h exch def /w exch def
   0 h rlineto w 0 rlineto
   0 h neg rlineto closepath stroke
} def
```

If the rectangles were to be filled rather than stroked, however, the first approach would work. (For more on transformations and the coordinate system, see the Red Book.)

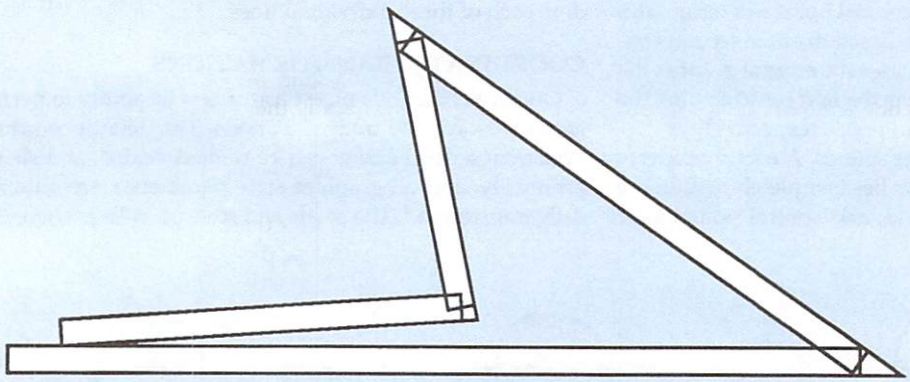


Figure 2: Stroke is handled by conversion to fill.

```
%! PostScript, but most structured comments removed for brevity.
%%BoundingBox: 70 70 410 210
newpath 80 80 moveto 400 80 lineto 230 200 lineto 250 100 lineto
100 90 lineto 10 setlinewidth strokepath 1 setlinewidth stroke showpage
```

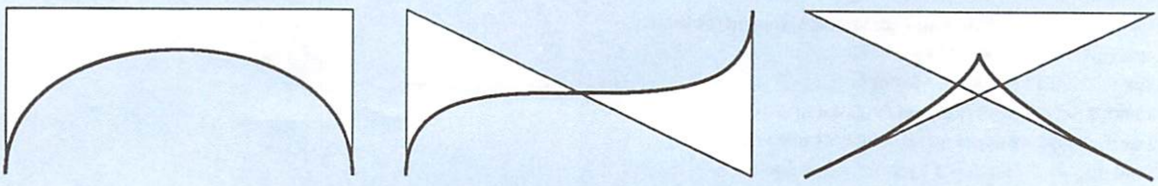


Figure 3: Bezier splines, with end points and control points.

```
%! PostScript, but most structured comments removed for brevity.
%%BoundingBox: 70 70 524 144
/c { 8 6 roll newpath gsave 1 index 1 index moveto
   0.25 setlinewidth 7 index 7 index lineto
   5 index 5 index lineto 3 index 3 index lineto
   stroke grestore newpath moveto curveto stroke } def
72 72 translate 10 5 10 67 140 67 140 5 c
150 0 translate 10 5 10 67 140 5 140 67 c
150 0 translate 10 5 140 67 10 67 140 5 c showpage
```

## RENDERING IN DEVICE SPACE

The resolution-independence and transformations available in PostScript mean that you cannot make assumptions about the resolution or device coordinate system. Each PostScript coordinate system unit might be 1/8 of a pixel or 1000 pixels. In addition, all of the points in a path might lie exactly on a device pixel or at any point between them.

PostScript considers device pixels to be perfect rectangles of a fixed size. Of course, printers do not draw perfect rectangles, but it is a useful approximation nonetheless. Remember that each graphics operation is broken down into a series of polygons that are filled with a specific color. When PostScript actually renders a polygon, any pixels that the polygon touches are filled, as Figure 4 illustrates.

Thus, a line of width zero in PostScript coordinates will generate a polygon of zero width running through the device pixels. The final rendered line will be one pixel wide in device space. A vertical line with a width equal to one device pixel generally touches two device pixels (because arbitrary user-space points almost never end up as integers in the device space) and thus generates a rendered line two pixels wide. An almost-vertical line with a width equal to half a device pixel will alternate between one and two device pixels wide. As an example, send the following lines to a 300-dot-per-inch (dpi) PostScript printer:

```
12 setlinewidth
72 72 moveto 78 288 lineto stroke
showpage
```

The result will look like a piece of twisted string as the width of the rendered line varies. Line widths of vertical and horizontal lines will vary according to their placement on the page, as well. A vertical line with a width of half a device pixel has an even chance of being drawn one or two pixels wide. This is a general problem with raster graphics, and there is no solution that does not introduce other problems.

Through PostScript's transform and itransform operators, programs do have access to the final device coordinate system. This can be used to snap each line endpoint to an exact device pixel, sacrificing a tiny bit of accuracy in the name of consistency. A common PostScript idiom, the sequence

```
transform round exch round exch itransform
```

moves a point given in user coordinates to the nearest device pixel boundary.

## COLOR AND GREY SCALES

The PostScript imaging model supports both grey scales and color, but you generally need not worry about how they are supported. We will only discuss grey scales here. (Color is supported in a similar fashion; each color component is dealt with separately.)

Most devices can only render two levels of grey—black and white. When rendering colors between white and black, PostScript uses dithering—an operation based on the one used to print newspaper photographs. A lattice of dots (known as a screen) is drawn, the dots spaced at equal intervals. The darker the grey, the larger the dots, until for black, they completely overlap. The spaces between the dots are also painted, but with an opaque white.

In general, for an ordered dither, an infinite lattice of small "tiles" is assumed to cover the page. Each tile is identical, and each consists of a fixed number of device pixels. A number between zero and one is associated with each pixel in a tile, and generally the numbers in a tile are fairly evenly spread over the range zero to one. Figure 5 illustrates this, with an example tile that might be the default in a 300 dpi PostScript printer.

If the current color is less than the number associated with the pixel, then a black pixel is drawn; otherwise, a white pixel is. With this procedure, a particular color turns a fixed number of pixels in each tile black and a fixed number white, yielding an approximate grey.

The number of greys that can be approximated is determined by the size of the tile. The size of the tile is determined by the spacing and orientation of the screen. Use the setscreen operator to adjust this. An angle of 45 degrees generally yields the best results for grey scales, and a screen spacing of 60 dots per inch (or lines per inch, in typesetting parlance) is the PostScript default.

Actually, because each tile needs to be the same size, arbitrary angles and resolutions are not possible. The PostScript interpreter performs the following functions in order

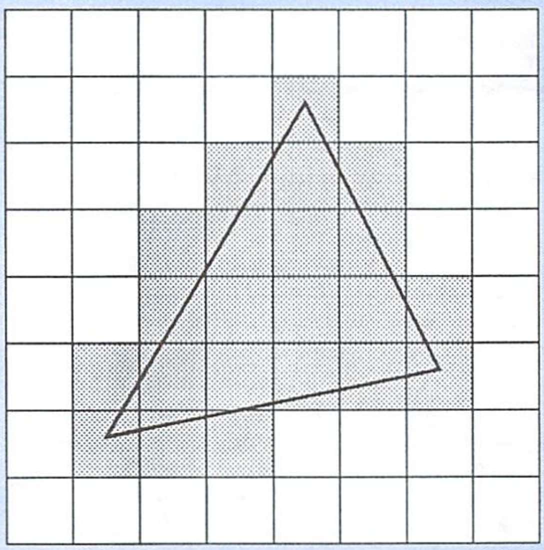


Figure 4: Painting in device space.

```
%! PostScript, but most structured comments removed for brevity.
%%BoundingBox: 70 70 324 324
/p { 25 mul } def /x 0 def
/r { p /x exch def p exch p exch moveto x 0 rlineto 0 1 p rlineto
   x neg 0 rlineto closepath fill } def
72 72 translate 1 setlinecap 0.25 setlinewidth
gsave 0.9 setgray 1 1 3 r 1 2 6 r 2 3 5 r 2 4 4 r 3 5 3 r 4 6 1 r grestore
0 1 8 { p /x exch def 0 x moveto 8 p x lineto stroke
    x 0 moveto x 8 p lineto stroke } for
1 setlinewidth
1.5 p 1.6 p moveto 6.5 p 2.6 p lineto 4.5 p 6.6 p lineto closepath stroke
showpage
```

to calculate the spot size:

```
x-off = round(device-resolution / screen-resolution * cos(screen-angle))
y-off = round(device-resolution / screen-resolution * sin(screen-angle))
tile-size-in-pixels = x-off * x-off + y-off * y-off
```

For a 300 dpi device and the default parameters, each offset is 4 and the tile size is 32 pixels, for a total of 33 possible grey scales. The actual frequency and resolution of the screen can be calculated from the above parameters by:

```
actual-screen-resolution = device-resolution / sqrt(tile-size-in-pixels)
actual-screen-angle = atan(y-off / x-off)
```

The actual screen for our example is 53 lines per inch at 45 degrees.

Note that because of the way a sample for each device pixel is filtered through the tile, the shape or position of the spots in the presence of a grey scale gradient can change, yielding a more accurate image. (This also happens with photographic screens.)

If a set of irregularly-shaped tiles can cover the plane, with all tiles oriented the same way, an equivalent set of rectangular tiles can also cover the plane, with a one-to-one correspondence between the points on the original tiles and those on the rectangular tiles. PostScript interpreters use this mathematical theorem to speed up rendering.

This tiling mechanism supports both spot dithers and ordered dithers, as well as output devices that are capable of more than two grey levels. It does not, however, support such error-diffusion dithers as Floyd-Steinberg. Properly supporting error-diffusion dithering in a PostScript device would significantly complicate and slow rendering, and no PostScript interpreter I am aware of has been extended in this fashion.

PostScript can also render general raster graphics with image and imagemask. These are rendered through a sampling process: Each pixel in device space touched by the image is mapped back to that image, and the color of the pixel at that position in the raster graphic is used with the screen tiles to set the device-space pixel black or white. No filtering is done; low-resolution images scaled up look blocky, and high-resolution images containing fine or high-resolution detail when scaled down suffer from aliasing artifacts.

Most PostScript interpreters can render images scaled to correspond one to one with device pixels very rapidly, so it is possible and not terribly inefficient to use a PostScript printer as a bitmapped device where necessary.

## CLIPPING

PostScript also contains the notion of a clipping path. Each polygon rendered is automatically clipped to the current clip path, which is initialized to the area on the page that can be rendered. A polygon is clipped by removing all portions of it that lie outside the polygons defined by the current clip path. The clip operation takes the current path and intersects it with the current clip path and makes the result the current clip path. The clippath operator returns the current clipping path. Thus, to draw the entire page with white, you could use

```
1 setgray clippath fill
```

With the clip path, you can perform many nice operations such as fountains (gradient fills) and irregularly-shaped raster images. Figure 6 offers an example.

## STATES AND RESOURCES

One of the best features of PostScript that is absent from many other graphics systems is the abstraction of graphics state. Each "global" parameter that affects rendering operations is considered part of that state. These parameters include the current transformation from user to device space, the line width, the line cap and join style, the current path, and the clip path. PostScript uses two operations to save and restore the graphics state—gsave and grestore. The former saves the current state on the state stack, leaving it unchanged. The latter pops the state stack and restores the graphics state to what it was at the time of the save.

The fact that the current path is considered part of the graphics state is very useful. One common paradigm is to fill a polygon with a light color and stroke it with a darker one. This requires both a stroke and a fill on the current path. The sequence:

```
gsave 0.5 setgray fill grestore 0 setgray stroke
```

does this operation nicely for any path.

A major benefit of using gsave and grestore is it saves the

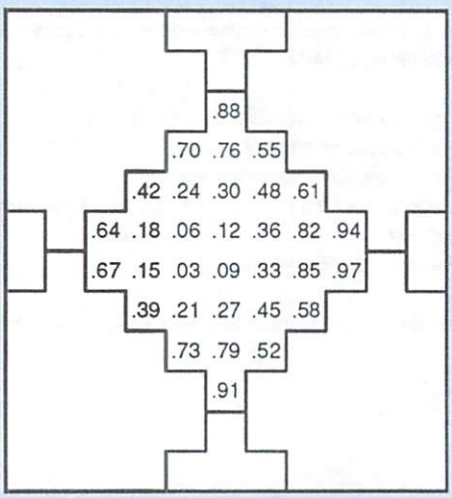



Figure 5: A halftone cell, with threshold values.

```
%! PostScript, but most structured comments removed for brevity.
%%BoundingBox: 70 70 239 254
72 72 translate 1 setlinecap /p 15 def /x 0 def /y 165 def
/Helvetica findfont 8 scalefont setfont /- { /x x p add def } def
/nr { /x 0 def /y y p sub def } def /l { x y moveto 0 p rlineto stroke } def
/b { x y moveto p 0 rlineto stroke } def
/n { dup stringwidth pop p exch sub 2 div x add 5 y add moveto show - } def
- - - - l b - - b - l nr - - - - - l b - l nr - - - - b - l (.88) n l b nr
- - - b - l (.70) n (.76) n (.55) n l b nr
b - - b - l (.42) n (.24) n (.30) n (.48) n (.61) n l b - - b nr
- l b - l (.64) n (.18) n (.06) n (.12) n (.36) n (.82) n (.94) n l b - l nr
b - l - l b (.67) n (.15) n (.03) n (.09) n (.33) n (.85) n b (.97) n l - l b nr
- - - l b (.39) n (.21) n (.27) n (.45) n b (.58) n l nr
- - - - l b (.73) n (.79) n b (.52) n l nr - - - - - l b (.91) n l nr
- - - - b - l - l b nr - - - - l - - - l 0 0 moveto p 11 mul 0 rlineto
0 p 12 mul rlineto p -11 mul 0 rlineto closepath stroke showpage
```

entire graphics state. Should more features be added to PostScript that would require global parameters to affect rendering, these parameters would become a part of the graphics state. Therefore, all PostScript programs using gsave and grestore would function correctly without change, even if they directly or indirectly called PostScript subroutines that use the new features. This is important for PostScript applications that import graphics from a variety of sources.

In addition to the ability to save the graphics state, PostScript has the ability to save virtually the entire interpreter state. The save operation takes the current interpreter state, including all definitions, collects it into a single save object, and pushes that object onto both the save stack and the operand stack. The restore operation takes a parameter that must be a save object created by a previous save, scans the save stack for a matching object, and restores the state of the interpreter from that object. All uses of def, array modifications, memory allocations, and other changes to the state of the interpreter are reversed or undone. The only exception is that changes to strings are persistent across save levels. The save operator includes an implicit gsave and restore does an implicit grestore. A save object can only be restored once, but at any time any pending save object can be restored. When a save object is restored, every save object created after that save object is invalidated.

Save and restore are PostScript's only mechanism for reclaiming memory: PostScript does no garbage collection. Memory is allocated by many operators, including string, array, and dict. In general, if you use any PostScript memory (which you will if you use PostScript), to be a good player, you are required to use save and restore to reclaim that memory. I'll illustrate the proper use of save and restore next time in Part II.

## FONTS

Because text is the most common type of graphics created, PostScript contains powerful and efficient mechanisms to support text. PostScript fonts are nothing more than a set of PostScript procedures inside a special dictionary—as a matter of fact, the dictionary itself is the font. There are a few entries in that dictionary that must exist and there are certain conventions that must be followed, but in general, a PostScript font can be any arbitrary PostScript program.

One of the most important features of PostScript is its generality regarding which fonts you can use. Using standard PostScript transformations, fonts can be stretched, rotated, flipped, written backwards, printed along a curve, and contorted many other ways. Drop shadows are child's play. No other page description language I am aware of has the capabilities of PostScript with regard to fonts.

There are two major types of PostScript fonts, Type 1 and Type 3. Type 1 fonts are fully-hinted outline fonts that take advantage of an excellent font renderer built into each PostScript interpreter. The format of these fonts is extremely restricted, but they are the only fonts that can really take advantage of "hints," which supply information about how a particular outline should be rendered at low resolutions. (For typesetting, anything below about 1200 dpi is low resolution; hints make a dramatic difference at 400 dpi and below.) For an in-depth discussion of fonts, consult "Font Formats" by Dan Weiss (p. 51, October '91).

Choose carefully: Many so-called PostScript interpreters do not handle Type 1 fonts or do not support the hints. On the Amiga, Post by Adrian Aylward (available on the October '91 disk) fully supports hinted Type 1 fonts.

Because of the highly restricted format of Type 1 fonts, it is possible to support them with much less code than a PostScript interpreter would require. As a matter of fact, Type 1 fonts are so restricted that they should really be considered data rather than programs.

Type 3 fonts are more general, as they are able to contain virtually any PostScript code, but they cannot themselves use hints. (They can, however, use a hinted Type 1 font, thus benefitting from hints for those portions rendered by the Type 1 font.) Most user-defined special-purpose fonts are Type 3. To support Type 3 fonts, you need a full PostScript interpreter.

In a PostScript font, all characters are accessed by name, not character code, so other character encodings or character sets can be defined at will. The Encoding entry in a font dictionary contains the mapping from character code to character

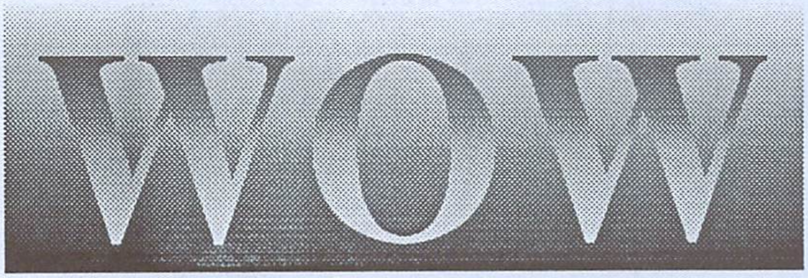



Figure 6: Using clipping to create fountains.

```
%! PostScript, but most structured comments removed for brevity.
%%BoundingBox: 70 70 374 174
/x 300 def /y 0 def /ypos 12 def /Times-Bold findfont 100 scalefont setfont
/str (WOW) def /xpos x str stringwidth pop sub 2 div def
72 72 translate 0 1 100 { /y exch def gsave
   newpath 0 y moveto x 0 rlineto 0 1 rlineto x neg 0 rlineto closepath clip
   y 100 div setgray clippath fill
   100 y sub 100 div setgray xpos ypos moveto str show grestore
} for showpage
```

name; the most common entry is StandardEncoding, which is an extended ASCII character set. It is relatively easy to remap a font by copying a font and then creating a new Encoding vector. In addition, a font may contain more than 256 characters, some of which would only be accessible through remapped versions of the font.

A normal text page contains about 2500 characters. Rendering each of these characters from outlines individually would normally take the PostScript interpreter a long time. Luckily, only 100 or so different characters are actually used on a given page. PostScript uses a font cache to dramatically improve the rendering time of fonts. The program that renders each character in a font can mark that character as eligible for the font cache (with setcachedevice). If this is done and the character is reasonably small, it is added to the font cache. The next time that character is used, if it exists in the font cache, the bits representing that character will be taken directly from the cache. The font cache generally uses a least-recently-used scheme to replace characters, and it is exempt from save/restore operations. The font cache improves the PostScript interpreter's speed at least tenfold.

### TUNE IN AGAIN

In this article, we examined the major features of PostScript and some details on how rendering is done. Next time, we will discuss the structured comment conventions required for portability, some limitations of PostScript, and some common programming errors. ■

*Tomas Rokicki is president of Radical Eye Software and author of AmigaTEX. Contact him at Box 2081, Stanford, CA 94309, on BIX (radical.eye) or as sysop of Radical Eye Radio (415/327-2346).*

## Bibliography

These books should be in any serious PostScript programmer's library. All are published by Addison-Wesley, and unless otherwise noted are by Adobe Systems Inc.

*Adobe Type 1 Font Format*, Version 1.1, ISBN 0-201-57044-0.

*PostScript Language Program Design*, ISBN 0-201-14396-8.

*PostScript Language Reference Manual*, Second Edition, ISBN 0-201-18127-4.

*PostScript Language Tutorial and Cookbook*, ISBN 0-201-10179-3.

*Thinking in PostScript*, By Glenn Reid, ISBN 0-201-52372-8.



STATEMENT OF OWNERSHIP, MANAGEMENT, AND CIRCULATION

1A. Title of publication: AmigaWorld Tech Journal. 1B. Publication no.: 10544631. 2. Date of filing: Sept. 18, 1991. 3. Frequency of issue: bimonthly. 3A. No. of issues published annually: 6 (reporting on 3 issues). 3B. Annual subscription price: $69.95. 4. Complete mailing address of known office of publication: 80 Elm Street, Peterborough, Hillsborough County, New Hampshire 03458. 5. Complete mailing address of headquarters of general business offices of the publisher: 80 Elm Street, Peterborough, Hillsborough County, New Hampshire 03458. 6. Full names and complete mailing address of publisher, editor, and managing editor: Publisher: Stephen Robbins, 80 Elm St., Peterborough, NH 03458. Editor: Linda Barrett Laflamme, 80 Elm St., Peterborough, NH 03458. Managing Editor: none at this time. 7. Owner: International Data Group, One Essex Plaza, Boston, MA 02116-2851. 8. Known bondholders, mortgagees, and other security holders owning or holding 1 percent or more of total amount of bonds, mortgages, or other securities: Patrick J. McGovern, One Essex Plaza, Boston, MA 02116-2851. 9. For completion by nonprofit organizations authorized to mail at special rates: not applicable. 10. Extent and nature of circulation: (X) Average no. copies each issue during preceding 12 months; (Y) Actual no. copies of single issue published nearest to filing date; A. Total no. copies (X) 15,999 (Y) 16,400. B. Paid and/or requested circulation: 1. Sales through dealers and carriers, street vendors, and counter sales: (X) 4,452 (Y) 4,583. 2. Mail subscription: (X) 2,381 (Y) 2448. C. Total paid and/or requested circulation: (X) 6,833 (Y) 7,031. D. Free distribution by mail, carrier, or other means (samples, complimentary, and other free copies): (X) 175 (Y) 75. E. Total distribution: (X) 7,008 (Y) 7,106. F. Copies not distributed: 1. Office use, left over, unaccounted, spoiled after printing: (X) 3,402 (Y) 3,692. 2. Return from news agents: (X) 5,589 (Y) 5,602. G. Total: (X) 15,999 (Y) 16,400.

# Advanced Use Of The Audio Device

*Learn how to make your Amiga sound off with more than a beep tone.*

**By David "Talin" Joiner and Joe Pearce**

YOU PROBABLY KNOW that audio.device provides a standard, system-friendly way to control the Amiga's sound-creating hardware. Before you say, "Oh, no. Another simple beep-tone tutorial," listen. We're going beyond the beep to show you how to make the audio device do actual, useful work that you can directly patch into your applications.

To give you a firm foundation before we jump into the code, consider some interesting facts about the audio device:

• The audio device is relatively quick. Most calls to it (at least the ones that do real work, such as playing a sound) can be implemented via BeginIO(), which provides a low-overhead, direct access into the driver. Also, despite the fact that the audio device is a very generalized mechanism, it is not as slow as you might think. At one time, we wrote our own specialized audio driver to get better performance. We found that, while there was some speed-up, it wasn't been worth the effort we put into it.

• You can call many audio.device functions directly from a vertical blank interrupt or a soft interrupt. This is useful if you are writing a music driver or some other system that initiates the playing of sounds in real time.

• Audio.device allows several commands to be queued in advance. This is useful if you are playing IFF samples, such as FORM 8SVX (the standard IFF sound-sample form). 8SVX samples occur in two parts: a "one-shot" part (which plays once at the beginning of the sound) and a "repeat" part (which repeats over and over). This is used mostly in the creation of musical instruments, for example a trumpet sound. The trumpet has an initial "blip" as the shockwave of air travels through the tube (which can be recorded in the one-shot part of the sound), followed by a continuous musical tone (which can be reproduced by recording a fragment of the tone as the repeat part of the sound, and then repeating it until the note ends).

With the audio device, playing back such a sound is very simple. You create two audio requests, one for the one-shot part and one for the repeat part. For the one-shot audio request, you set the number of repeats to 1, because you only want the section to play once. For the repeat part, you set the number of repeats to 0, which tells the audio device that you want it to play that portion an infinite number of times. Then send both audio requests to audio.device using BeginIO(). After the one-shot part has finished, the audio device will automatically play the repeat part, without any intervention on behalf of your program.

You can stop the note at any time by sending a CMD_FLUSH to the audio device. This will stop any notes that are playing on a particular audio channel and remove all pending requests from the queue. That way, if the one-shot part is still playing, CMD_FLUSH will not only stop it, but also prevent the repeat part from playing. CMD_FLUSH can also be used to flush all audio channels as well as a single one; you give it a bitmask telling it what channels you want it to affect. An alternate way to stop the note is to call AbortIO() for the individual audio requests, as shown in the example code in the accompanying disk's Joiner drawer.

• The audio device is easily shared. For example, Joe has a script that runs in the background on his system, and every hour it speaks the time using the narrator device (which, obviously, uses audio.device). In addition, we have other programs that beep or generate sounds that we are working on. Because we have written all of our code to take advantage of audio.device's ability to be shared, he can still hear his script speaking the time, even when we are running our other programs.

Try to keep the audio device channels free for other programs to use as much as possible. In a beep-type program, I only allocate a channel just before the beep and free it up when the beep is done. Similarly, in a game with music playing in the background, when the last note of the song is finished, I free all the channels so that some other program can use them. As far as keeping audio.device open, it doesn't matter which way you do it—it doesn't cost anything to keep it open, and it opens fairly quickly.

### THE POOL STRATEGY

One thing you will soon discover about the audio device is that you need a lot of IOAudio structures. Nearly everything is done with an IOAudio or IORequest, and because most events happen asynchronously, it's sometimes tricky to tell when a particular IORequest is free for use. If you are doing music, for each active channel you'll need one IOAudio for the one-shot part and one for the repeat part. In addition, you'll need a global IOAudio for the CMD_FLUSH, which, fortunately, is synchronous.

I set up a "pool" of IOAudio structures that are all hanging off a message port. I create the pool by setting up an anonymous, private message port, called PoolPort, and setting its flags to PA_NONE, so that no signal is sent when it receives a message. Then I create a whole bunch (about ten) of IOAudio structures. I set the reply port of each IOAudio's message to PoolPort and then reply to all of them. This caus- ►

es all the IOAudio structures to be attached to PoolPort.

Now, whenever I need an IOAudio structure, I call GetMsg() on PoolPort, and it gives me the next available structure, if there is one. I fill in that structure with the required variables and send it to the audio device. After the sound finishes playing, audio.device will call ReplyMsg() for the IOAudio structure. Because the reply port is still set to PoolPort, the structure will automatically return to the pool of messages, ready to be used again.

## AUDIO.DEVICE IN ACTION

Illustrating some of audio.device's features mentioned above, play (in the Joiner drawer of the accompanying disk) is a CLI-only program that plays an IFF 8SVX sampled sound. The command-line syntax is as follows:

**play [FILE] <filename> [PERIOD nnn] [OCTAVE n] [STEREO]**

The FILE parameter (the word FILE is optional) specifies the name of the sampled sound file to play. PERIOD and OCTAVE allow you to select the value of the hardware period register and which octave in the sample file to play, respectively. STEREO is a flag that indicates that the player should simultaneously play the sound on both the left and right channels.

If the sample has a repeat part, it will continue to play until you hit Control-C. The main routine parses the command-line parameters and reads the samples sound file into memory, calling the function Read8SVX(). This function is a fairly standard IFF reader that fills in a struct SampleData defined in the supplied include file. Note, however, that we could have just as easily used IFFParse.

*"The Amiga audio hardware is tied to the system's color clock."*

You might notice the use of the dreaded goto inside the ERROR macro. In this case, the gotos are being used in a very strict manner, as a type of "extended return" statement. The macro ERROR(x) is defined to mean "Return with an error, but before you do, run the cleanup code for this function." So in a sense we are extending the language in a way that makes it more readable, rather than creating spaghetti code. (My apologies to the language purists out there.)

After the sample is loaded, we print out any author or copyright information that the sample might have. We then start playing the sound. At this point, we go into a busy loop waiting for either a Control-C or the sample to finish playing. Note that this is not a good example of multitasking, and it's not intended to be.

After we break out of the loop, we stop the sound, deallocate all of the resources we grabbed, and exit.

Now, let's take a look at the actual routine that plays a sample. First we build ourselves some IOAudio structures—as few as one (if we are playing in mono for a sample that has only a one-shot part) or as many as four (if we are playing in stereo a sound that has a one-shot part and a repeat part). Next, we open audio.device and allocate the channels that we'll need, again depending on whether we are playing in mono or stereo. Now we fill in the structures that we just created. We use the IOF_QUICK flag to indicate that we want the routine to return immediately, regardless of its success. If we're playing in stereo, we make a second set of structures as well for the other channel. Note also that the code handles the one-shot part and the repeat part separately . Finally, we send all of these requests to the audio device, using BeginIO().

The CheckSample() routine simply uses the CheckIO() system call to determine if the IORequest is finished. Note that we have to be careful, because there might be more than one IORequest pending, and we want to make sure that we check not only the one-shot part, but also the repeat part, if there is one.

StopSample() uses AbortIO() for the sake of simplicity in the example, although CMD_FLUSH or a number of other ways to stop the sound could be used as well. Again, we have to be careful, because there could be as many as four IORequests floating around. After we abort the IORequests, we need to WaitIO() on them, because AbortIO() does not always happen immediately. Even when it does, we need to remove the IORequests from the message port, and WaitIO() does this for us quite nicely. The last thing StopSample() does is to free up everything we allocated.

The Read8SVX() routine starts by opening the IFF file, looks at the FORM type to make sure everything's kosher, and then goes into a large loop, looking for chunks. Each chunk it encounters, it either reads (if it recognizes it) or skips over (if it doesn't). When it gets to the BODY chunk, it performs an additional calculation to determine the system's color-clock speed. The Amiga audio hardware is tied to the system's color clock, meaning that samples will play at just a slightly different frequency on a PAL machine than on a NTSC machine. This calculation compensates for the difference. Note that the calculation is much more efficient under the 2.0 operating system, because more information is available about that sort of thing.

Finally, FreeSampleData() frees all the resources that have been allocated in the process of reading the file.

## SOUND EXERCISES

If you were going to use code like this in a real application, you would obviously need to make a few changes. For example, in most applications you'll want to wait for several different kinds of events at once, such as IntuiMessages, signals, and so on. The audio device code presented here can be integrated into such an architecture in a fairly straightforward fashion, as long as you have a fairly solid understanding of message ports, IORequests, and signals. If you don't, I suggest playing around with them and experimenting until you do—these three mechanisms are basic to almost everything on the Amiga, and a thorough understanding of them will get you a long way towards understanding everything else. ■

*David "Talin" Joiner is the author of Music-X and Faery Tale Adventure, plus an artist, award-winning costume designer, and moderator of the user.interface topic of the Amiga.sw BIX conference. Joe Pearce is author of Discovery 2.0 and Commodore's Installer utility. Contact them c/o* The AmigaWorld Tech Journal, *80 Elm St. Peterborough, NH 03458, or on BIX (talin).*

SAVE $29.75

# A source of technical information for the serious Amiga professional.

Introducing *The AmigaWorld Tech Journal*, the new source to turn to for the advanced technical information you crave.

Whether you're a programmer or a developer of software or hardware, you simply can't find a more useful publication than this. Each big, bi-monthly issue is packed with fresh, authoritative strategies to help you fuel the power of your computing.

Trying to get better results from your BASIC compiler? Looking for good Public Domain programming tools on the networks and bulletin boards? Like to keep current on Commodore's new standards? Want to dig deeper into your operating system and even write your own libraries? Then *The AmigaWorld Tech Journal* is for you!

Our authors are programmers themselves, seasoned professionals who rank among the Amiga community's foremost experts. You'll benefit from their knowledge and insight on C, BASIC, Assembly, Modula-2, ARexx and the operating system—in addition to advanced video, MIDI, speech and lots more.

Sure, other programming publications may include some technical information, but none devote every single page to heavyweight techniques, hard-core tutorials, invaluable reviews, listings and utilities as we do.


*Every issue includes a valuable companion disk!*

And only *The AmigaWorld Tech Journal* boasts a technical advisory board composed of industry peers. Indeed, our articles undergo a scrupulous editing and screening process. So you can rest assured our contents are not only accurate, but completely up-to-date as well.

PLUS! Each issue comes with a valuable companion disk, including executable code, source code and the required libraries for all our program examples—plus the recommended PD utilities, demos of new commercial tools and other helpful surprises. These disks will save you the time, money and hassle of downloading PD utilities, typing in exhaustive listings, tracking down errors or making phone calls to on-line networks.

The AmigaWorld

TECH JOURNAL

In every issue of *The AmigaWorld Tech Journal*, you'll find…

- Practical hardware and software reviews, including detailed comparisons, benchmark results and specs.
- Step-by-step, high-end tutorials on such topics as porting your work to 2.0, debugging, using SMPTE time code, etc.
- The latest in graphics programming, featuring algorithms and techniques for texture mapping, hidden-line removal and more.
- TNT (tips, news and tools), a column covering commercial software, books and talk on the networks.
- Programming utilities from PD disks, bulletin board systems and networks.
- Wise buys in new products—from language system upgrades to accelerator boards to editing systems and more.

The fact is, there's no other publication like *The AmigaWorld Tech Journal* available. It's all the tips and techniques you need. All in one single source. So subscribe now and get the most out of your Amiga programming. Get six fact-filled issues. And six jam-packed disks.
Call 1-800-343-0728 or complete and return the savings form below—today!

*To order, use this handy savings form.*

☐ Yes! Enter my one-year (6 issues, plus 6 valuable disks) subscription to *The AmigaWorld Tech Journal* for the special price of $59.95. That's a special saving of $29.75 off the single-copy price. If at any time I'm not satisfied with *The AmigaWorld Tech Journal*, I'm entitled to receive a full refund — no questions asked!

Name ______________________

Address ______________________

City ____________ State ________ Zip ________

☐ Check or money order enclosed. ☐ Charge my:

☐ MasterCard ☐ Visa ☐ Discover ☐ American Express

Account No. ______________ Exp. Date ________

Signature ______________________

*Satisfaction Guaranteed!*

*Or your money back!*

Canada $79.95 (includes GST)
Mexico $74.95
Foreign surface $84.95
Foreign airmail $99.95
California residents add $3.75 tax
Payment required in US funds drawn on US bank.

Complete and mail to:

*The AmigaWorld Tech Journal*
PO Box 802, 80 Elm Street
Peterborough, NH 03458

4CSB2

*For faster service, call toll-free 1-800-343-0728.*

ON DISK

# Better Bezier Curves

*Irregular does not mean impossible. With these routines, your program can produce smooth, nonuniform, curvilinear shapes.*

By Robert Wittner

CAD, STRUCTURED-DRAWING, and 3-D modeling programs all need to represent nonlinear objects. The rasterization algorithms for circles, ellipses, arcs, and the like are fairly straightforward. For nonuniform, curvilinear shapes, however, we must resort to slightly more complicated methods. The possible implementations range from Bezier curves to nonrational uniform B-splines to Catmull-Rom curves. Of all of the available choices, the Bezier curve is probably the most popular and simplest to implement. As you will see, you can easily support them in your own programs.

## SEMANTICS

The Bezier curve is a form of a cubic polynomial curve segment. (See Figure 1.) Each curve segment has a starting point, an ending point, and two control points. The control points are used to control the shape of the curve between the start and end points. The curve is tangent to the vector P0P1 at the start of the curve and tangent to P2P3 at the end of the curve. The vector magnitude is also used to represent the "velocity" of the curve at each end point. The slope of the curve at each point between the end points is interpolated from the slopes derived from the control points.

You can derive a two-dimensional Bezier curve from ($x_0$, $y_0$) to ($x_1$, $y_1$) using ($x_2$, $y_2$) and ($x_3$, $y_3$) as control points from the following pair of cubic equations:

$$x(t) = a_x t^3 + b_x t^2 + c_x t + x_0$$
$$y(t) = a_y t^3 + b_y t^2 + c_y t + y_0$$

These equations are defined as t ranges from 0 to 1. The supplementary equations for determining the a, b, and c coefficients are:

$$x_1 = x_0 + c_x/3 \qquad y_1 = y_0 + c_y/3$$
$$x_2 = x_1 + (c_x + b_x)/3 \qquad y_2 = y_1 + (c_y + b_y)/3$$
$$x_3 = x_0 + c_x + b_x + a_x \qquad y_3 = y_0 + c_y + b_y + a_y$$

## SOLUTIONS

There are several methods for implementing the rasterization Bezier curves. The least complicated (and the one I will discuss and optimize) is successive linear approximation. You simply loop t from 0 to 1 in an orderly fashion and draw lines from each point you calculate to the next. Other methods include recursive subdivision and the forward differential. The recursive-subdivision approach involves subdividing the curve into halves until you reach the desired approximation accuracy, at which time you draw the curve via connected line segments. In the forward-differential method you take derivatives of the parametric functions and apply the derivative functions to calculate points along the curve (for line-segment drawing). This technique eliminates some multiplication during each point's calculation, however, the setup required is complex and prone to error.

The setup for the linear method requires values for $a_x$, $a_y$, $b_x$, $b_y$, $c_x$, and $c_y$. These can be derived by algebraic manipulation of the supplementary equations as follows:

$$c_x = 3(x_1 - x_0) \qquad c_y = 3(y_1 - y_0)$$
$$b_x = 3(x_2 - x_1) - c_x \qquad b_y = 3(y_2 - y_1) - c_y$$
$$a_x = x_3 - x_0 - c_x - b_x \qquad a_y = y_3 - y_0 - c_y - b_y$$

Once you calculate these values, you can begin to loop t from 0 to 1 and calculate each point. However, the parametric equations given above can be rewritten more efficiently. Using Horner's rule to factor the parametric equations, you can modify the original equations to:

$$x(t) = ((a_x t + b_x)t + c_x)t + x_0$$
$$y(t) = ((a_y t + b_y)t + c_y)t + y_0$$

This factoring reduces the number of multiplications for each curve point by two.

With the mathematics completed, it is fairly easy to implement a working C subroutine to draw a Bezier curve, given a RastPort and a partially initialized Border structure. The examples below and in the Wittner drawer on disk use Intuition drawing routines for generality, but you can easily replace them with calls to graphics.library functions for greater speed.

```
BOOL DrawBezier(struct Border *B, struct RastPort *R, int x0, y0, x1,
                                        y1, x2, y2, x3, y3, USHORT NumSegs)
{
    float t;
    SHORT Cur[4], ax, bx, cx, ay, by, cy;

    /* Only drawing one segment at a time */
    B->Count = 1;
    B->XY = Cur;

    /* Calculate intermediate values */
    cx = (x1 - x0) * 3;
    bx = ((x2 - x1) * 3) - cx;
    ax = x3 - x0 - cx - bx;

    cy = (y1 - y0) * 3;
    by = ((y2 - y1) * 3) - cy;
    ay = y3 - y0 - cy - by;


    /* Main loop to draw curve */
```

```
    for (t = 0; t <= 1; t += (1 / NumSegs))
        {
            Cur[2] = ((ax * t + bx) * t + cx) * t + x;
            Cur[3] = ((ay * t + by) * t + cy) * t * y;

            if (t == 0)
                {
                    /* Initialization for first time through */
                    Cur[0] = Cur[2];
                    Cur[1] = Cur[3];
                }

            /* Draw a segment */
            DrawBorder(R, B, 0L, 0L);

            /* Move end point of this line to start point of next line */
            Cur[0] = Cur[2];
            Cur[1] = Cur[3];
        }
}
```

## EXTENSIONS

Some extensions you should consider are integer arithmetic (increasing speed), multiple Bezier curves (increasing image complexity), and filling Bezier curves. Integer arithmetic involves using integer variables to hold fixed-point, fractional information. The assumption is made that each integer variable contains a number n such that n divided by some constant amount gives the true value of n. For example, if "Total" is an integer variable, and the constant multiplier is 10000, then to find out the true value of Total, you divide Total by 10000 and examine the results. This technique is used frequently to speed up mathematical calculations when accuracy is not critical. Because basic integer math is native to the main processor, the processing time drops dramatically at the expense of accuracy. Because there is only a limited, fixed amount of accuracy, rounding errors can occur. An example of an integer Bezier curve calculation is in the Wittner drawer of the accompanying disk.

For most real-world applications, one Bezier curve will not be sufficiently complex to define a particular shape. You must resort to chaining multiple curves together to form more complicated shapes. Different configurations are possible depending on how the curve segments are placed adjacent to one another.

If the end point of one curve and the start point of another coincide and the second and first control points (respectively) are collinear with the end points, the curve will appear smooth from one segment to the next. If the control points are not collinear, the curve will appear to have a kink or sharp bend at the connecting point of the two segments.

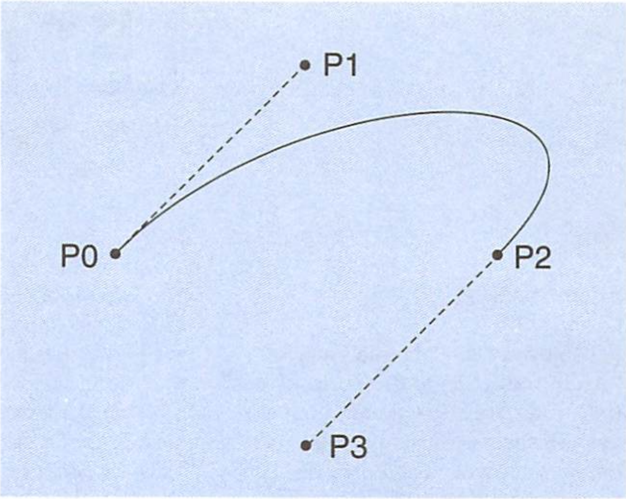



**Figure 1: A Bezier curve with the start point P0, end point P2, and control points P1 and P3.**

For a closed shape to be completely smooth, the last curve segment must join the first in the manner described above. If the curve is drawn this way, it will produce the best results when filling the curves as described below.

The Amiga's polygon-filling graphics primitives make filling curves straightforward. Use the AreaMove(), AreaDraw(), and AreaEnd() functions to draw small polygon segments. Allow the Amiga to do its scan conversion on these segments, and then draw them. The only differences between this routine and the simple routine given above are:

- You must compute line segments for more than one curve.
- You must allocate space to store all the segment coordinates. (Alloc=Word_Size * 2 * #_of_Curves * #_of_Segs_per_Curve)
- You must use the Area...() commands to draw the curve segments.

Study the sample routine for drawing a filled Bezier shape in the Wittner drawer to see the mechanics of this technique.

For further study, an excellent source of computer graphics information is *Computer Graphics – Principles and Practice* by Foley, van Dam, Feiner, and Hughes (Addison-Wesley, 1990). The book covers many aspects of computer graphics, such as simple 2-D primitive drawing (line, circle, ellipse, polygon), area filling, photorealistic rendering (ray tracing, scanline algorithms, radiosity algorithm), solid modeling, fractal systems, particle systems, and computer animation. Some of the math presented in this article was derived from the information on Bezier curves given in this book and in the *Postscript Language Reference Manual* by Adobe Systems Inc. (Addison-Wesley, 1985–88). Otherwise known as the "Red Book" or the "Postscript Bible," the *Reference Manual* covers all aspects of using Postscript to generate printed material. It also contains a brief, but useful, section on Bezier curves.

If you discover any further optimizations, please let me know! ■

*Robert Wittner is a Programmer Analyst for Rockwell International, and is working on the Space Station Freedom program. He has been programming the Amiga for four years. You can contact him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

# LETTERS

*Flames, suggestions, and cheers from readers.*

### SPEC SEARCH

I enjoyed Dave Haynie's article, "An Introduction to the Zorro III Bus" (p. 2, June/July '91). So far, I have been unsuccessful in my attempts to obtain a copy of "The Zorro III Bus Specification." Would you give me further information on this document? Thank you.

**Richard Olson, Jr.**
*Huntsville, Alabama*

*CATS decided not to sell the specification as a stand-alone document to the general public. You can find the information, however, in Appendix K of the* Amiga Hardware Reference Manual, Third Edition *(ISBN: 0-201-56776-8). Published by Addison-Wesley (Route 128, Reading, MA 01867, 617/944-3700), the book retails for $28.95 ($37.95 in Canada).*

### DISK DILEMMA

Please stop putting the right and left parentheses in your disk directory names. Maybe these work fine on some machines, but I have serveral problems using them on my machine with 1.3 and ARP.

Also, I have little use for the .info files. They are useless for any .lzh files. Clicking on a compressed file's icon just flashes the screen. You might as well leave the icons off, except for executables and text files.

**Victor Ross Parkerson**
*Las Cruces, New Mexico*

*Victor, yours was one of a flood of complaints about the parentheses. You will be pleased when you look at this issue's disk. From now on, we no longer use parentheses in directory names. We hope this makes the disk easier to navigate.*

*For now, the icons for .lzh files will stay. Before we added these we received calls and letters from readers who, after double-clicking on the drawers, decided their disks were "empty." The icons are intended as reassurance for readers who like to make a quick visual survey of a disk's contents. If space becomes tight, however, unessential icons will be the first to go.*

### LIBRARIES AND DEVICES

I am very happy with *The AmigaWorld Tech Journal*'s level of technical content, but I have two suggestions.

First, the Journal could specify needed shared libraries that are useful to developers. Some examples are libraries for hash tables (hash.library), DES (des.library), b trees (btree.library), ent trees (enttree.library), and signal processing (sigproc.library). The library definitions should include a .fd file, a driver file for testing, and tips for how to proceed. You could also include information about existing libraries that are not written up in the Amiga documentation. This way, Amiga programmers can better help each other and avoid covering the same ground.

Second, I hope you provide information on how to program Amiga devices. I would like to see a morse code device that converts the incoming file to morse code (either through the speakers or out one of the back ports to HAM equipment).

**Gregg Myer**
*Lawrence, Kansas*

*In case you missed it Gregg, check out the December '91 issue. The disk contains 13 libraries, plus the code for "Designing a Device Driver" found on page 44 of that issue. Along the same lines, the Applications drawer of this issue's disk is packed with utilities to make programmers' lives easier.*

*We're always on the lookout for tools and libraries to include on disk and welcome your suggestions on what's new and useful or submissions of your own timesavers.*

### LOST WORLDS

I've searched everywhere on Internet and I can't find the WorldMap directory for World DataBank II mentioned in "Global Parlor Tricks" (p.22, October '91). Where is it?

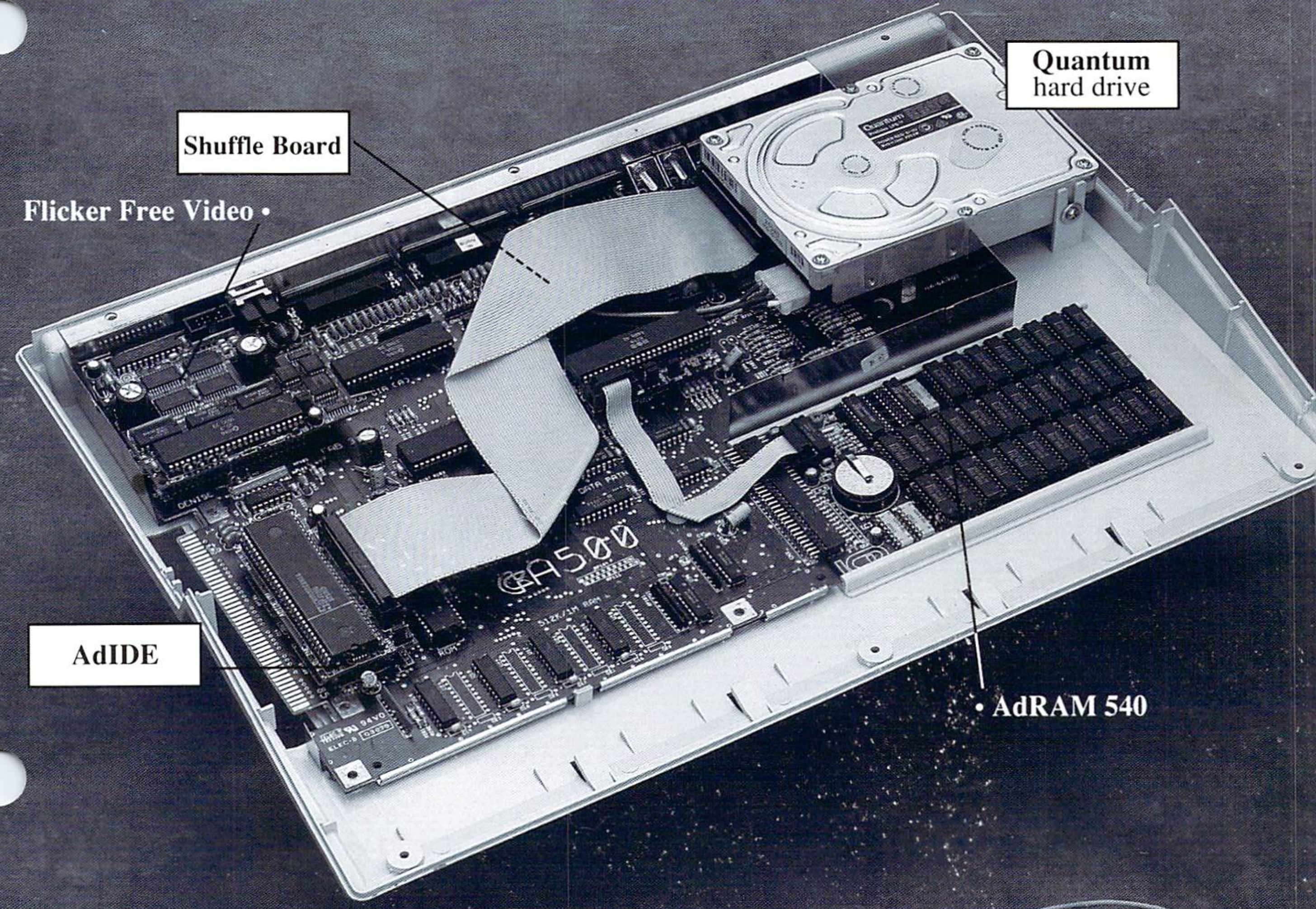
**Ray Wallace**
*Kalispell, Montana*

*You can't find it, because it's not there. Unbeknownst to Howard Anderson, the article's author, the database was posted to a "volatile" area of Internet; data posted there is deleted after four days. To obtain a copy of World DataBank II, contact Howard at his Internet address (anderson@atc.sps.mot.com). He would be happy to post a copy for you, or anyone else who needs it... just be sure to retrieve it within four days!*

### LET US KNOW

*What are your suggestions, complaints, and hints for the magazine, Amiga developers, and your fellow readers? Tell us about them by writing to Letters to the Editor,* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or posting messages in the AW.Tech Journal conference on BIX. Letters and messages may be edited for space and clarity.* ■

# Prima!
## A Look Inside the Ultimate A500.




ICD proudly presents **Prima**™.. the high performance, low cost hard drive for Amiga® 500 computers. Prima blends a large capacity, low power Quantum™ hard drive with the **AdIDE**™ host adapter for an unbeatable combination.

**Prima** replaces the internal floppy drive but includes **Shuffle Board**™ to make your external floppy drive DF0:. **Prima** features auto–booting from FastFileSystem partitions, high speed caching, auto–configuring, and A–MaxII™ support. Formatted capacities of 52 and 105 megabytes are currently available.

**Prima** comes complete with instructions, software, and all the hardware necessary for a simple, clean, no–solder installation. It does require an A500 with switching power supply, 1 megabyte of RAM, and an external floppy drive for setup and installation.

What other products would we include in the "Ultimate A500"? Of course a four megabyte **AdRAM**™ **540** and **Flicker Free Video**™ with a multi–sync monitor. Why settle for less?

ICD

ICD, Incorporated
1220 Rock Street
Rockford, Illinois 61101
USA (815) 968-2228 Phone (800) 373-7700 Orders (815) 968-6888 FAX



Circle 2 on Reader Service card.